Model-based Testing of Operating System-Level Security Mechanisms

Yakoub Nemouchi

NNT : 2016SACLS061

THESE DE DOCTORAT
DE
L'UNIVERSITE PARIS-SACLAY
PREPAREE A
" L'UNIVERSITE PARIS-SUD"

ECOLE DOCTORALE N° 508
Science et Technologies de l'Information et de la Communication

Spécialité de doctorat : Informatique

Par

M. Yakoub NEMOUCHI

Model-Based Testing of Operating System-Level Security Mechanisms

Thèse présentée et soutenue à Orsay, le 30/03/2016 :

Composition du Jury :

M. Poizat Pascal	Professeur Université Paris Ouest Nanterre	Président
M. Hierons Robert	Professeur Brunel University	Rapporteur
M. Merz Stephan	Directeur de Recherche INRIA Nancy	Rapporteur
M. Boulanger Frédéric	Professeur CentraleSupélec	Examinateur
M. Wolff Burkhart	Professeur Université Paris-Sud	Directeur de thèse

Model-Based Testing of Operating System-level Security Mechanisms

Yakoub Nemouchi

14th April 2016

Contents

Abstract

Formal methods can be understood as the art of applying mathematical reasoning to the modeling, analysis and verification of computer systems. Three main verification approaches can be distinguished: verification based on deductive proofs, model checking and model-based testing.

Model-based testing, in particular in its radical form of theorem proving-based testing [BW13], bridges seamlessly the gap between the theory, the formal model, and the implementation of a system. Actually, theorem proving based testing techniques offer a possibility to directly interact with "real" systems: via different formal properties, tests can be derived and executed on the system under test. Suitably supported, the entire process can fully automated.

The purpose of this thesis is to create a model-based sequence testing environment for both sequential and concurrent programs. First a generic testing theory based on monads is presented, which is independent of any concrete program or computer system. It turns out that it is still expressive enough to cover all common system behaviours and testing concepts. In particular, we consider here: sequential executions, concurrent executions, synchronised executions, executions with abort. On the conceptual side, it brings notions like test refinements, abstract test cases, concrete test cases, test oracles, test scenarios, test data, test drivers, conformance relations and coverage criteria into one theoretical and practical framework.

In this framework, both behavioural refinement rules and symbolic execution rules are developed for the generic case and then refined and used for specific complex systems. As an application, we will instantiate our framework by an existing sequential model of a microprocessor called VAMP developed during the Verisoft-Project. For the concurrent case, we will use our framework to model and test the IPC API of a real industrial operating system called PikeOS.

Our framework is implemented in Isabelle/HOL. Thus, our approach directly benefits from the existing models, tools, and formal proofs in this system.

Part I

Introduction and Context

Introduction

Contents

1.1 Motivations

Formal verification techniques, i. e. a family of methods that establish the correctness of programs wrt. a specification, have seen a remarkable boost in recent years. In particular methods based on deductive code-verification or model-checking can, within the boundaries of certain foundational assumptions, provide an *absolute* guarantee in a sense for the correctness of programs in a system. It is safe to state that the formal verification of computer systems becomes increasingly relevant due to the critical roles they play in daily human life, and this is reflected by their role in certification processes assuring that certain security or safety requirements are respected by a wider and wider range of products. An example of a critical systems, an embedded system comprising safety critical software components, e. g. the engine controller of an aeroplane.

However, pursuing our thought experiment a little further, we have to admit that flying our just-formally-verified aeroplane would simply be illegal, and for good reasons: The approval of an aeroplane (and other safety critical systems) is legally bound to formal certification process (such as DO178B/C[SR10, Bro11], Common criteria [Mem06], etc.), which requires a combination of *tests* and verification techniques. One reason is that deduction based verification may establish the correctness of code towards a specification, i. e. a mathematical model, but this doesn't guarantee that the model and its foundational assumptions correspond in sufficient precision to the physical reality in the embedded system. Another reason is that existing certification standards simply did not consider the possibilities of modern formal verification techniques, and certification engineers do not know what guarantees can and cannot be gained from the use of formal methods. Thus, a plane should never fly without verifying both, the code and its underlying modeling assumptions of its critical components, by a combination of test and proof techniques, enforced by an adequate formal certification process.

While certifications of large systems, including fully functional operating systems up to Common Criteria EAL[1] 4 [Com06] are common practice today, higher levels involve the use of formal methods in terms of combined test and proof activities, covering various layers of a system including soft and hardware-components. To reach EAL7, one has to formally specify a *security policy model* (called SPM), a model of the system operations called *functional specification model* (the FSP), and a kind of refinement proof between these two. Finally, a battery of tests have to be provided that establish the correspondence between the FSP and the "real" implementation in form of code. One of our goals is to contribute to the test-effort for an EAL 5 or higher certification of PikeOS [2] operating system by a test-method designed to support this activity. All three, modeling, certification and test effort were pursued the European EURO-MILS [3] project aiming at an EAL 5 certification for PikeOS, were the organizational context of this work.

At present, specification-level verification and the development of test sets are usually two unrelated tasks. While test sets for certification kits are usually developed manually and independently from the specification, our model-based test case generation approach , developed during this thesis, uses a design model that can already be used for the verification task. Beyond the advantage of natural integration of our model-based testing techniques into a certification process, model-based testing using symbolic evaluation can treat models with complex state, which distinguishes it from popular model checking techniques [Mer01], and connect a model with a real implementation, without additional assumptions (i. e. correctness of the com-

[1] Evaluation Assurance Level

[2] www.pikeos.com

[3] www.euromils.eu

piler, existence of the model of the underlying hardware, etc.). The latter distinguishes our approach from classical deductive verification techniques [ABGR10].

In formal testing, a branch of model-based testing, the properties of systems are verified by a testing experience based on a formal model. The goal of a test is to establish a *conformance relation* between the model and a system, which is a kind of a satisfaction relation that must, for practical reasons, be based on a *finite test set*, i. e. containing a generated set of test cases.

In our view, it is not possible to treat formal models only by paper and pencil notations as is the case in many works and publications. Models and proofs treated in this work i. e. the system model of a real operating system, its complex operational semantics and the resulting very complex symbolic execution rules, developed by a collaborating team comprising several persons routinely changing basic definitions, is out of reach of conventional paper and pencil theory development; a proof assistant for routinely re-checking definitions and re-proving proofs is indispensable in such a development effort. Thus, we will use interactive theorem proof assistant to carry our *testing framework*.

Interactive theorem proving is a technology of fundamental importance for mathematics and computer-science. It is based on expressive logical foundations and implemented in a highly trustable way. In this thesis, our approach is implemented inside the interactive theorem proving environment Isabelle/HOL extended by a plugin with test generation facilities called HOL-TestGen. The use of Isabelle/HOL as a modeling environment has the following advantages:

1. We inherit all its technical features, e. g., formal modeling and verification, code generation and document generation,
2. Our test case generation algorithm is based on the symbolic computation engine implemented as Isabelle tactics. Thus, can count as highly trustworthy,
3. HOL-TestGen allows us to seamlessly integrate formal verification and testing in a unique way.

1.2 Contributions

The main focus of our work is to provide a theorem proving-based sequence testing environment for both sequential and concurrent complex systems[4].

[4]In this document we will use the term complex systems to designate a computer program with complex component, e. g. the kernel of an OS

In fact, this kind of test environments is advantageous during different certification processes. We will divide the list of our contributions into three groups.
Our theoretical contributions consist in:

- the extension of the monad-based test framework, introduced in [BW13], by new concepts, e. g., executions with abort, interleaving, to express sequence test scenarios for concurrent and sequential programs,
- the derivation of behavioral refinement rules and symbolic execution rules for the new introduced concepts,
- the embedding of the standard test refinement in the monad testing framework, and
- a conformance relation based on observed error codes, to test security properties, e. g. information flow and access control, that links the specification and the implementation using abstract test drivers.

On the technical side we would like to mention:

- the definition of key theories to test computer systems, e. g. a theory on shared memory,
- the introduction of an optimized scheme to derive symbolic execution rules and an efficient way for their implementation in Isabelle/HOL, and
- a proposal to build test drivers that links abstract tests derived on Isabelle level with concrete concurrent code.

On the methodological side, our contributions are:

- a guideline to convert a functional system model to a testable theory, gained from a substantial case study,
- a methodology to control the execution of a concurrent program during our test experience, and
- a high-level mandatory guidelines and recommendations for both developers and evaluators of certification documents containing Isabelle specifications.

In fact, our contribution can be seen as a proposal of a test and proof environment composed of a tool chain Figure 1.1 that goes from the abstract Isabelle/HOL level down to code-level (e. g. C, SML). First, we will introduce a monad based sequence testing theory encoded in Isabelle. It is not

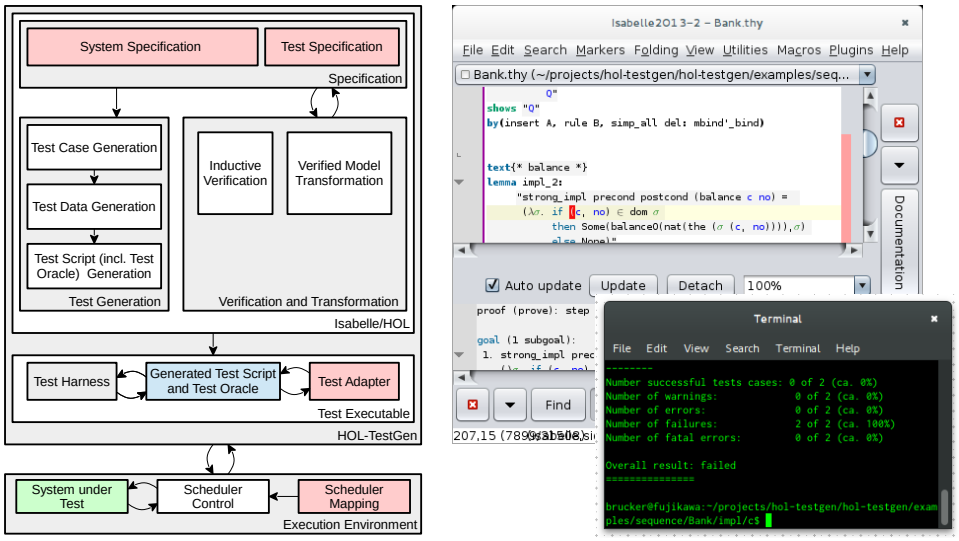


Figure 1.1: The HOL-TestGen Workflow.

restricted to a particular computer system, but we believe that monads are expressive enough to cover all common behavioral concepts (e. g. sequential executions, synchronized executions, etc.) and testing concepts (test oracles, test scenarios, coverage criteria, etc.).

Then the proposed testing theory is extended by a non-standard behavioral concepts (i. e. aborted executions, concurrent executions with abort) and test concepts (i. e. a new coverage criterion to test IPC protocol, error-codes based conformance relation) to express security and functional test scenarios for systems executed in a concurrent context. Moreover, the functional model of both, the operating system PikeOS and the VAMP processor, are embedded in our monadic framework, and a test experience for the two systems was established.

In this thesis, we will also develop a test case generation process that fully relies on a symbolic execution rules established as Isabelle lemmas, i. e. formally derived rules. In fact, each system under test has its own specific operational semantic, which means that, the generic symbolic execution rules on the introduced monads operators are not optimized enough to execute any operational semantic. However, a refined versions of the generic symbolic execution rules are derived for the different case studies presented in this thesis. Moreover, and using Isabelle/ML which is a development environment for ML programming offered by Isabelle, we will develop tactics that help into the automation of the process of the application of symbolic execution rules on a given test scenario designed inside our framework.

Our contribution also covers, the implementation of test drivers to test concrete code. A test driver is composed from three main components: the test

script, the adapter and the test harness. While HOL-TestGen code generator, which is actually a refined version of Isabelle code generator, is used to generate automatically test scripts in SML language, two programs implemented in Isabelle/ML will be used as a test harness and a test adapter for the test driver. Actually, and in order to test C concurrent code, we will add another program to our standard construction of test drivers. The program is also implemented in Isabelle/ML, and it uses the test script to generate a set of gdb files. In fact, during our test experiences, we will show how gdb can be used to control the executions of a given concurrent program implemented in C language. Thus, we avoid putting strong assumptions related to the non-determinism of the system scheduler choices during a concurrent execution. Finally, we will use `MLton` compiler as an interface to connect our test scripts written in SML level with implementations in C code-level, and build our test executables.

1.3 Overview

The idea behind this thesis is to design a test and proof environment for sequential and concurrent complex systems. In order to present this work, we have organised the document as following:

Part I: Introduction and Context

In the first part of the document we will introduce the context of this thesis. Formal methods, in particular the use of a verification technique based on formal testing approach implemented in a theorem proving environment, is the topic of this thesis. The chapter 2 contain a description of formal methods and its relation with formal testing. The chapter also contains a description for the formal development environment used in this thesis which is Isabelle/**hol!**. Moreover, the architectures of two systems used in our case studies is presented in this chapter.

Part II: Contributions

The second part of the document contains our contributions during this thesis. This part is devided into four chapters:

chapter 3: Isabelle in Certification Processes

This chapter was published as an internal technical report [YABC15]. The chapter introduce mandatory recommendations for the evaluators of CC documents containing Isabelle theories. Actually it is an instantiation of Eric

Jaeger text[JH08], a document that contain recommendations for evaluators of CC documents.

chapter 4: Theoretical and Technical Foundations

The content of this chapter was partially published in [BHNW15a]. In this chapter, we will introduce our test framework. Our framework is represented by a tool-chain consisting of:

1. **The Specification Language**: used to express different behaviors of computer programs. The specification language is based on a monad theory formalized on a top of Isabelle/**hol!**.

2. **HOL-TestGen**: it is an extension of Isabelle with test case generation facilities, e. g. trace generator based on Isabelle datatypes package, an interface to connect an Isabelle local proof context to constraint-solvers, etc.

3. **Test Drivers**: they are programs used to execute automatically the generated tests on a given program under test written in a given programming language, in particular we consider programs implemented in: `sml`, `OCaml`, `scala`, `haskell`, `C` and `F#`. In our approach, test drivers represent a link between the model and the program under test. Their implementation is not fully automatic. While some parts of the test driver is generated automatically, e. g. the test script, a particular part of it, which is the test adapter, is written by hand.

chapter 5: Testing VAMP Processor

The content of this chapter is published in [BFNW13]. In order to meet requirements of a certification process for critical security systems, one has to formally verify properties on the specification as well as test the implementation thoroughly. This includes tests of the used hardware platform underlying a proof architecture to be certified. To this end, in this chapter we present a case study for the model-based generation of test programs (i.e, the basis for a certification kit) for a realistic model of a RISC processor called VAMP. In this case study we use an existing model of VAMP and HOL-TestGen to develop several conformance test scenarios.

chapter 6: Testing PikeOS API

A part of this chapter was published in [BHNW15a]. The chapter introduces another case study for model-based test generation, but this time, our investigation covers the software layer, more precisely the API of an industrial concurrent embedded system. The chapter introduces a model of PikeOS

embedded in our "monadic" test theory. That covers an extension of the theory to embed interleaving executions with abort, synchronization, and shared memory. Experiments on the IPC API and their results are also the topic of this chapter.

chapter 7: Conclusions

Finally, we sum-up with different achievements of this thesis and our future works related to the topic.

Context

Contents

2.1 Introduction

Formal methods describe a set of mathematically based techniques and tools for specification, analysis and verification of computer systems. They are mainly used to describe and to verify, in a logically consistent way, some properties of these systems.

During a formal testing activity, a well-established branch of formal methods, the properties of systems are verified through a testing experience based on a satisfaction relation between the formal model (the specification) and the implementation of a system. More precisely, the goal of a test experience is to establish a *conformance relation* between a model and an implementation, that must, for practical reasons, be based on a *finite* test set. Consequently, testing attempts to run the real system and attempts to establish a *verdict* on a necessarily finite set of *observations*. The obvious fundamental limitations of the testing approach can be partly overcome by the following techniques:

1. test sets can be generated and the generation procedure can be designed to generate a different test set for regeneration. So a sequence of a decently organised regression test can increase the confidence in a software development process well enough.

2. the generation of test sets can be guided by a well-chosen *coverage criteria* whose effectiveness can be established by empirical observations in a concrete software development process.

From our point of view, formal testing is a sub-field of Model-Based Testing (MBT) since often semi-formal languages (e. g. UML[RJB99]) were used to generate tests. Because test suites are derived from models and not from source code, both formal testing and model-based testing are usually seen as a form of black-box testing. In the context of our work, our *testing theory* is developed in a formal environment, called the *testing framework*, implemented inside an interactive theorem proving environment.

Interactive theorem proving (ITP) is a technology of fundamental importance for mathematics and computer-science. Applications include very large mathematical proofs and semi-automated verification of complex software systems. ITP systems are based on expressive logical foundations and implemented usually in a highly trustable way; this is due to the architecture of contemporary ITP systems such as Coq [Wie06, §4], Isabelle [NPW02] or the HOL family [Wie06, §1] (HOL4[Kum13], HOL light [Har09], etc.) going back to the influential LCF system [MW79] from 1979, which has pioneered key principles like correctness by construction for primitive inferences and definitions, free programmability in user-space via `SML`, and top-level command interaction.

The purpose of this chapter is primarily to present preliminaries on formal

testing and on its specific branch: *sequence testing*. Moreover, we will bring together a body of system information that is generally known in the Isabelle community, but largely scattered in system documentations and papers. This includes a brief introduction into the system, a general overview over the methodology and covers certain aspects of the tool support. Such an introduction into Isabelle and its higher order logic implementation will help the reader to understand the different concepts and approaches proposed in this thesis. In fact, the general context of this thesis is: the *implementation* of a *test and proof environment* that relies on *Isabelle*, with the intention of using it during a *certification processes* of critical systems. Actually, during our investigations, our testing theory was extended by two substantial case studies, namely a micro-processor called VAMP[BJK+06] and a real-time operating system PikeOS[1][SYS13a]. Some concepts related to the latter are also presented in this chapter.

The chapter proceeds as follows: A general introduction into formal testing followed by an overview on sequence testing are presented in section 2.2 and subsection 2.2.2. In section 2.3, we provide a guided tour over the Isabelle system. In subsection 2.3.7 we describe an extension of Isabelle used for model-based testing, and discuss its advantages and limits in model based testing area. In section 2.4 and section 2.5, the basic design concepts and the system architectures related to VAMP and PikeOS are presented.

2.2 Formal Testing and Prover-Based Testing

The relation between deductive proof verification, model checking and formal testing is complementary and fruitful in our view, although the three approaches use common formal specifications techniques during the verification process. This relation is sketched in Figure 2.1. The advantages of using formal specifications by the three approaches can be summarized as follow:

- A formal specification language provides a mathematically precise notation to express properties of systems.
- In software engineering, formalizing the syntax and the semantics of specification languages leverages tools for automated reasoning on systems.
- Formal specifications of systems can systematically be refined to code [PS83, Abr96].
- In critical systems, specifications based on interactive theorem-proving tools can be used to prove that an implementation is free of bugs

[1] PikeOS is a brand-name of SYSGO AG

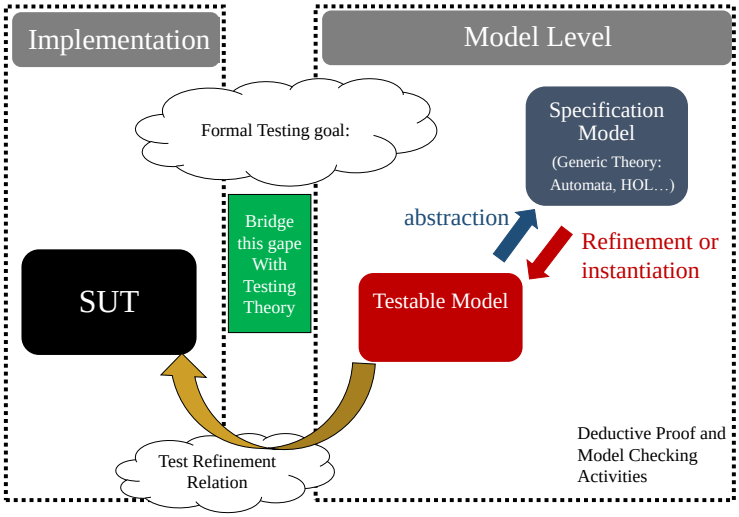


Figure 2.1: A Test and Proof Framework

and that it satisfies its formal specification in every possible execution [Bar03, CDH+09].

- In testing activity, test cases can be generated automatically from formal specification [Car81, Gau95, BKM02]
- Model-Based test techniques can compile a given formal specification to oracles that determine when a particular test passes or fails [HBB+09] .

From a conceptual point of view, a testing activity can be seen as the establishment of a conformance relation between the model and the system under test in order to meet a given test requirements. In fact, in our Prover-Based Test (PBT) approach, the conformance relation is expressed within a formal testing framework. From our view, a formal testing framework is composed from:

1. the *test specification*: it is a higher order logic formula that express a property for the generated tests. In our framework a test scenario for a given test experience is formally expressed by a test specification. According to the test requirements, and based on the definitions inside the test theory, a test specification is designed under some *testing hypothesis* and *coverage criteria* in a form of, a *test refinement relation*. The latter contains a category of *conformance relations* that link the model with the real system. The test specification is designed in the logical context of a background theory, called in our terminology the

test theory. The test theory relies on *testability hypotheses* and contains concepts to express the test requirements for a given model-based test experience. For example a test theory contains: (a) a specification language to express the behavior of the system under test and also a formalization the properties to be checked during the test experience. Note that in our approach, the specification language, also called the model, has to be *testable*, i. e. can be refined to code, in order to be used for experiments, (b) defintions of concepts needed for the establishment of a given test experience, e. g. *conformance relations*, coverage criteria, test scenarios etc., and (c) the test strategy, e. g. symbolic execution process, data selection process, test execution process etc.

2. The scenario is technically captured by a *test suite*, which is a kind of a container comprising: (a) *test theorem*, (b) *abstract tests*, (c) *concrete tests* and (d) *other data related to a test scenario*,

3. the concrete test cases are *executed* on the system under test and the *test results* are derived, and

4. finally, a *verdict* is established, i. e., the test results are *interpreted* and some conclusions related to the test experience are stated. Of course, during the evaluation of the test experience, both testability assumptions and test results are used.

Different testing approaches and techniques and several ways of their integration inside a formal context were explored. In our context, we choose PBT approach, it has been applied to unit testing [BW09, BW13, BFNW13, BBW15], and during our work PBT was also applied to sequence test scenarios [BFNW13, BHNW15b].

2.2.1 On Theorem Proving Based Testing (PBT)

The idea of using a test-generation method based on theorem proving environment is particularly attractive for establishing the link between the model and the real implementation. In recent years, HOL-TestGen [BW13] has been developed for testing models presented in HOL, in particular for operations with complex data-structures, so data-types comprising lists, sets, trees, records, etc. Tests were generated in the logical context of a background theory and wrt. to a particularly property (called *test-specification*) formulated in it. At the begin of this thesis, HOL-TestGen was mostly geared towards the generation of unit-tests and test-specifications of the form:

$$\mathrm{pre}(x) \rightarrow \mathrm{post}(x, \mathrm{SUT}(x))$$

where x is arbitrary input, so possibly also containing an input state, pre and post condition, and SUT an uninterpreted constant symbol representing the system under test. The test-specification schema covers test scenarios where the initial state of the system is known and the result state is returned by the SUT; it is therefore assumed to be accessible in principle. HOL-TestGen provides automatic procedures to decompose via data-type splitting rules and a kind of DNF-normalization the initial test-specification into abstract test cases, i. e. clauses containing SUT(x) plus a collection of logical constraints on x. For example in [BW13], the authors want to express the property "SUT is a sorting algorithm on integer lists" by the test specification:

```
1    sort (list) = SUT (list)
```

where sort has been specified by, for example, an insertion-sort. A test case generation could yield the test cases in the (complete) test theorem:

```
1        1. [] = SUT []
2        2. THYP ([] = SUT [] ⟶[] = SUT [])
3        3. [x] = SUT [x]
4        4. THYP ((∃x. [x] = SUT [x]) ⟶(∀x. [x] = SUT [x]))
5        5. PO (x < xa)
6        6. [x, xa] = SUT [x, xa]
7        7. THYP ((∃x xa. xa < x ∧[xa, x] = SUT [xa, x]) ⟶
8                (∀x xa. xa < x ⟶[xa, x] = SUT [xa, x]))
9        8. PO (¬ x < xa)
10       9. [xa, x] = SUT [x, xa]
11          (...)
12       20. PO ((x < xa ∧xb < xa) ∧¬xb < x)
13       21. [x, xb, xa] = SUT [xb, x, xa]
```

where the test hypotheses were marked by `THYP`, and the constraints on the variables inside the list are marked by `PO`. If these constraints are satisfiable, a constraint-solver can produce a ground instance for `x`, say `c`, and isolate `post(c, SUT(c))` as concrete test, if these constraints are unsatisfiable they are *infeasible* tests that represent impossible (empty) abstract test-cases. Eliminating infeasible test cases as early as possible is primordial for effective test generation; it is also the key advantage over random-based testing which tends to be hopelessly inefficient if pre-conditions are non-trivial. Finally, HOL-TestGen offers the possibility to convert concrete test suites via code-generators into test drivers in a variety of target languages.

HOL-TestGen and its methodology is an instance of *model-based testing* (see [ABC+13] for a recent survey over the field, which was pioneered by M.C. Gaudel at the beginning of the 90ies[GB91, Gau95]). However, its meth-

odology coined "proof-based testing" distinguishes itself from main-stream approaches by the following features:

1. rather than residing on small, decidable data-type theories in a propositional or first-order logic setting, HOL-TestGen embraces higher-order logic (HOL) and favors for background theories and test specifications abstract and concise mathematical descriptions rather than indirect problem-encoding;
2. HOL-TestGen allows for *instrumenting* the generation processes of abstract and concrete test cases by *derived rules*, i. e. rules that are short-cuts for the normalization and data selection phases which were justified by formal proof;
3. HOL-TestGen leverages the possibility to "massage" of a *given* model into a *testable* one; beyond aforementioned instrumentation of the process, an initial model can be *refined* or *restricted* to a model that is more suited for test-generation and its underlying needs for a symbolic execution process;
4. HOL-TestGen offers the possibility of a semantically controlled, clean integration from models to the test driver generation.

Prior work [BBW15] with HOL-TestGen had shown that sequence test scenarios could be treated effectively in principle, if the background theory is geared towards efficient symbolic execution and if the process is decently supported by automated reasoning. However, there is no direct way to generalize the reification technique used in [BBW15] to the PiKeOS model, something that will be tackled by this thesis. In our context, we are particularly interested in sequence tests, which we describe in sequel in more details.

2.2.2 A Gentle Introduction to: Sequence Testing

Sequence testing is a well-established branch of formal testing theory having its roots in automata theory. In formal testing, the model, also called the specification, and the system under test (SUT), also called the implementation, are usually belonging to different worlds (e. g. the specification is a logic based, and the SUT is implemented on C-level). The link between the two worlds is established by the refinement relation expressed on the model-level and complemented by *methodological assumptions*.

Methodological Assumptions

The methodological assumptions, sometimes called *testability hypotheses* in the literature, are used to bridge the gap between the model and the system

under test [BGM91, DY96]. An example on testability hypothesis is the test refinement relation, it states that the system under test is a refinement of the model. The main testability assumptions in sequence testing theory are summarized as follows:

1. The tester can reset the system under test (the *SUT*) into a known initial state,

2. the tester can stimulate the SUT only via the *operation-calls* and *input* of a known interface; while the internal state of the SUT is hidden to the tester, the SUT is assumed to be *only* controlled by these stimuli, and

3. the SUT behaves deterministic with respect to an observed sequence of input-output pairs (it is *input-output deterministic*).

The latter two assumptions assure the reproducibility of test executions. The latter condition does *not* imply that the SUT is deterministic: for a given input ι, and in a given state σ, SUT may non-deterministically choose between the successor states σ' and σ'', provided that the corresponding outputs (o', σ') and (o'', σ'') are distinguishable. Thus, a SUT may behave non-deterministically, but must make its internal decisions observable by appropriate output. In other words, the relation between a sequence of input-output pairs and the resulting system state *must be a function*. There is a substantial body of theoretical work replacing the latter testability hypothesis by weaker or alternative ones (and avoiding the strict alternates of input and output, and adding asynchronous communication between tester and SUT, or adding some notion of time), but most practical approaches do assume it as we do throughout this thesis. Moreover note, that there are approaches (including our own paper [BFNW13]) that allow at least a limited form of access to the final (internal) state of the SUT.

Following [CG07], testability hypothesis are fundamental to establish the proof of the conformance relation between the model and the system under test. In [TPHS10] the authors mention that "testing can never be complete: testing can only show the presence of errors, not their absence", which is a famous aphorism of Dijkstra. An answer to this statement was given by [Fel12], when the author mention that formal *exhaustive* testing can be used to show the correctness i. e. the absence of bugs if the testability hypothesis are satisfied. Since the exhaustive set of tests is generally infinite, other assumptions, called *testing hypothesis* are needed to complete the proof of the correctness.

Testing hypothesis

In [BGM91, Gau95] two fundamental testing hypothesis, called *uniformity and regularity hypothesis* were introduced. They have been improved and embedded in Higher Order Logic (**hol!**) by [BW13]. The latter mention that regularity hypotheses can be used to address the problem of test case generation for universally quantified variables ranging over recursive datatypes such as lists and trees. The author formalized the assumption by the following natural deduction rule:

$$\frac{\begin{array}{c}\big[|x| < k\big]_x \\ \vdots \\ Px\end{array}}{Py} \tag{2.1}$$

The rule express that P is always true if, it is true for all data x less than a given depth k. On the other hand side, uniformity assumption is used to bound the set of possible instantiations of a quantified variable, which is usually infinite. The assumption is formalized by the following logical formula:

$$(\exists\ x_1\ \ldots\ x_n\ .\ \ P\ x_1\ \ldots\ x_n) \longrightarrow (\forall\ x_1\ \ldots\ x_n\ .\ \ P\ x_1\ \ldots\ x_n) \tag{2.2}$$

This formula denote that if P is a true for a given instantiation x_n then it is true for all instatiations of the type of the variable x. During our work, we will consider the latter testing hypothesis in connection with *coverage criteria*.

Coverage Criteria

The concept of coverage is mandatory in testing theory whenever exhaustive tests are impossible. If not *all* cases can be tested, a test coverage question can be raised. The question that must be answered is, did we *test enough*? For instance, if the test experience "fails", i. e. does not reveal any bugs under a given coverage, the latter can show to the tester where he can test more. The set of test cases must contain one test sequence for each executable path in the SUT can be seen as an example of a coverage for a given test experience. In [SLZ07] five interesting coverage criteria based on concurrency fault models were introduced. We will adapt, refine and formalize some of these criteria in **hol!** to test concurrent code inspired by PikeOS.

The Conformance Relation

A conformance relation is a satisfaction relation between a specification and a system under test, for which we assume it behaves like a function. A

conformance relation can be expressed by, e. g. equality, bisimulation, etc. Some conformance relations between a system specification and a SUT are proposed in subsection 2.2.3.

Verdicts

In general, two possible interpretations (verdicts) for the test set are distinguished, the test can *pass* or *fail*. If the SUT behaves correctly wrt. the specification by satisfying the established conformance relation then we say that the SUT passes the test with success. On the other hand, if the SUT does not satisfy the conformance relation we say that the SUT fails to pass the test.

In the next sections, we will present the known formal models used for sequence testing activities and discuss their techniques. From our point of view, a formal model is usually oriented towards a description of *data* and *states* composed thereof, or *behavior* in the sense of a set of system traces. In some cases, a model can also describe timing as well as performance. Thus we distinguish the following testing models (specifications) categories:

- Testing approaches based on behavioral models: describe the system by the relationships between states (data). Such a relationships typically describe the associations between system operations (inputs) and the system state, e. g. Kripke structures or Process Algebras.
- Testing approaches based on data abstraction: data abstraction describe the behavior of a system independently of its implementation. For instance, the Input-Output relation is expressed by a function that should preserve a set of properties. The properties on the function are expressed by logical formulas. e. g. Axiomatic Specifications or Prover-Based Testing.

2.2.3 Background on Sequence Testing Models

Specification models possesses syntax and semantics for expressing sophisticated behavioral aspects of systems such as synchronisation and concurrency. In the sequel we will highlight some of these specification languages, in particular these are: Input Output Automata (IOA), Axiomatic Specifications.

Background

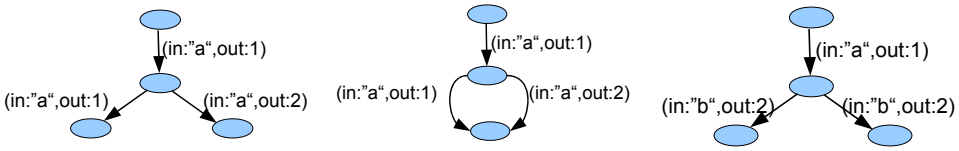
Specification languages provide a formal system annotations such as pre post conditions and *invariants* that allow to express the intended behavior of the system. Such specifications are useful precisely in development of computer systems. When used in conjunction with automated analysis and system

verification tools, such specifications can support detection of common vulnerabilities, generation of test cases and test oracles, and formal program verification. In the rest of this section we will first introduce theories related to sequence testing and then focus on specification languages that support concurrency.

Actually, Kripke structures as semantic basis of LTL-like languages, have been widely used as a formal specification formalism and in testing activities. One of the first works that introduces testing techniques on a Kripke-like structure was the experiments done by Moore [Moo56] on Finite State Machines (FSM). The idea of the experiments was based on interactions with a sequential machine in order to describe its behavior with a transition system. Inspired by fault detection experiments for sequential circuits represented by an FSM, Hennine [Hen64] introduced two testing concepts during his work. The first one is called *checking sequences*, which are a generated input sequences (from a source FSM i. e. the specification) that start from a given initial state. The checking sequences were executed on a target FSM in order to check that the execution of the sequence of inputs by the latter correspond to the execution of the source FSM by the same sequence. In our terminology the checking sequences can be seen as test cases and the satisfaction of the checking sequence by the target FSM can be seen as a kind of conformance relation. The second concept introduced by Hennine is *distinguishing sequence*. The concept of distinguishing sequence assumes basically that each input sequence starting from a given initial state is bound to an output sequence and the latter is different from all others generated from a different initial state. In our terminology this can be seen as a testability hypotheses. Based on the concepts introduced by Moore and Hennine, other testing theories equipped with new notions, and more complex Kripke structures were developed. For instance, Lee and Yannakasis in their work[LY94, DY96, LM96] discussed the use of distinguishing sequences and *Unique Input Ouput sequences* (UIO) to detect a non observable initial states. Actually, a lot of testing concepts and works were introduced for testing Finite State Machines, for more details on the story of testing theories, we would mention the remarkable background introduced by Feliachi [Fel12] in his Ph.D thesis and also the following surveys related to this topic [CSCS94, DY96, HBB+09, Gau10]. In the rest of this section we would like to focus on testing approaches that consider concurrent executions, since one of our contributions belong to the latter field.

IO-Automata Based Testing

An Input/Output Automaton is an automaton with finite number of states where each transition is represented by an alternation of a single occurrence of input or output events. A sequence of input-output pairs through an



(a) IO-Deterministic SUT. (b) IO-Deterministic SUT. (c) Non-IO-Determin. SUT.

Figure 2.2: IO-Determinism and Non-IO-Determinism

automaton A is called a *trace*, the set of traces is written $Trace(A)$. The function In returns for each trace the set of inputs for which A is enabled after this trace; in 2.2c for example, In $[(\text{“}a\text{”}, 1)]$ is just $\{\text{“}b\text{”}\}$. Dually, Out yields for a trace t and input $\iota \in In(t)$ the set of outputs for which A is enabled after t; in 2.2b for example, $Out([(\text{“}a\text{”}, 1)], \text{“}a\text{”})$ this is just $\{1, 2\}$.

Many approaches to test concurrent systems based on IOA were explored [BHJJ08, EH08, En-13]. In his work, Bochmann [BHJJ08] proposed a concurrent testing approach based on a new IOA model called Partial Order Input Output Automata (POIOA). A POIOA is a refined model of Multi-Port Automaton [LDvB+93], in which concurrency between inputs as well as inputs ordering constraints are considered. The idea behind this work is to define order constraints for inputs, and then based on this order, a set of test cases in a form of checking sequences is derived. Several conformance relations were proposed by the authors, in general the proposed conformance relations are based on the fact that the implementation must provide the same inputs outputs alternation (or a *quasi-equivalent* one) as the one proposed by the checking sequence derived from the specification. In other words, a trace T is quasi-equivalent to a trace T' if either $T = T'$ or T is obtained by reducing the input order constraints of T' (input-input or input-output), and/or T is obtained by increasing the output-output constraints. Other conformance relations between a specification given as automaton SPEC labelled with input-output pairs and a system under test are introduced in the literature:

- *input/output conformance (IOCO)* [Tre08b]: for all traces $t \in Traces$(SPEC) and all $\iota \in In(t)$, the observed output of SUT must be in $Out(t, \iota)$,
- *inclusion conformance* [PHL12]: all traces in SPEC must be possible in SUT and,
- *deadlock conformance* [FGWW13]: for all traces $t \in Traces$(SPEC) and $b \notin In(t)$, b must be refused by SUT

Testing Based on axiomatic specifications

The most of approaches allowing the derivation of test cases from a specification are based on behavioral descriptions of the SUT, for example:

- IO-Automa [LT89b].
- Control Flow Graph[All70a, All70b] of a given Program.
- Labeled Transition Systems [Tre08a].

Axiomatic specifications, also called algebraic specifications [BCFG86], are different. Actually, the specification of a system is represented by a a signature $\Sigma = (S, F, V)$ composed from a finite set of types S and a finite set of function names F and a set of variables V. The requirement during a test process based on algebraic specifications is, the satisfaction of the axioms or their consequences, defined on the functions in F by the SUT. In fact, this is different from the approaches adopted by behavioral oriented specifications, where the satisfaction relation is based on the possibility or impossibility of manifesting a given behavior by the SUT. Basically, a test case is an instantiation of the axioms, or their consequences, by the terms(functions and variables) of the SUT. The conformance relation is represented by the satisfaction of the axioms defined in the specification by the terms of SUT.

Many test theories based on algebraic specifications were developed [BCFG86, BGM91, DGM93, GLG08]. In the latter works, the authors expressed testability hypotheses as well as several exhaustive test definitions. Moreover note, testing hypotheses were proposed to deal with the problem of infinite test set. As examples of tools used to express algebraic specifications we mention: CASL[MHST08], ACT-ONE[EFH83] and OBJ [GMH81].

2.3 Isabelle/HOL

In the context of certifications of critical hard- and software systems, an understanding of its architecture and the underlying methodology may help to understand why Isabelle, if correctly used, can be trusted to a significantly higher extent than conventional software, even more than other automated theorem proving environments (in fact, Sascha Böhme's work on proof reconstruction [BW10] inside Isabelle revealed errors the SMT solver Z3[dMB08] that is perhaps the most tested conventional system currently on the market ...). Of course, Isabelle as software "contains errors". However, its architecture is designed to exclude that errors allow to infer logically false statements, and methodology may help to exclude that correctly inferred logical statements are just logical artifacts, or logically trivial statements, which can be impressing stunts without any value.

2.3.1 The Isabelle System Architecture

We will describe the layers of the system architecture bottom-up one by one, following the diagram Figure 2.3.

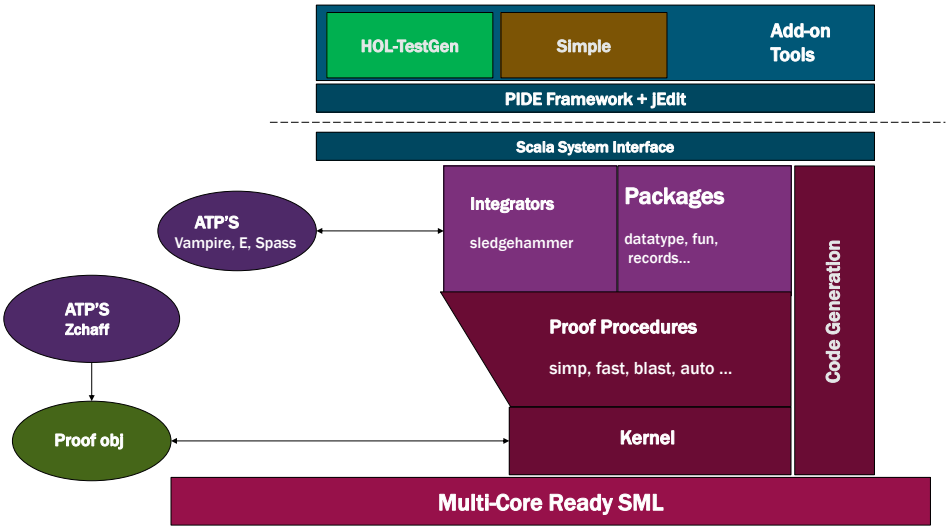


Figure 2.3: The diagram shows the different layers like execution environment, kernel, tactical level and proof-procedures, component level (providing external prover integration like Z3, specification components, and facilities like the code generator, the Scala API to the system bridging to the JVM-World, and the Prover-IDE (PIDE) layer allowing for asynchronous proof and document checking.

The foundation of system architecture is still the Standard ML (SML,[MTM97]) programming environment; the default PolyML implementation www.polyml.org supports nowadays multi-core hardware which is heavily used in recent versions for parallel and asynchronous proof checking when editing Isabelle theories.

On top of this, the logical kernel is implemented which comprises type-checking, term-implementations and the management of global contexts (keeping, among many other things, signature information and basic logical axioms). The kernel provides the abstract data-types `thm`, which is essentially the triple (Γ, Θ, ϕ), written $\Gamma \vdash_\Theta \phi$, where Γ is a list of *meta-level assumptions*, Θ the *global context*, containing, for example, the signature and core axioms of `HOL` and the signature of group operators, and a *conclusion* ϕ, i. e.a formula that is established to be derivable in this context (Γ, Θ). Intuitively, a `thm` of the form $\Gamma \vdash_\Theta \phi$ is stating that the kernel certifies that ϕ has been derived in context Θ from the assumptions Γ.

There are only a few operations in the kernel that can establish `thm`'s, and

the system correctness depends *only* on this trusted kernel. On demand, these operations can also log proof-objects that can be checked, in principle, independently from Isabelle; in contrast to systems like Coq, proof objects do play a less central role for proof checking which just resides on the inductive construction of `thm`'s by kernel inferences shown, for example, in [MW10].

On the next layer, proof procedures were implemented - advanced tactical procedures that search for proofs based on higher-order rewriting like `simp`, tableau provers such as `fast`, `blast`, or `metis` , and combined procedures such as `auto`. Constructed proofs were always checked by the inference kernel.

The next layer provides major components — traditionally called *packages* — that implement the *specification constructs* such as *type abbreviations*, *type definitions*, etc., as discussed in subsection 2.3.3 in more details. Packages may also yield connectors to external provers (be it via the `sledgehammer` interface or via the `smt` interface to solvers such as Z3), machinery for (semi-trusted) code-generators as well as the Isar-engine that supports structured-declarative and imperative "apply style" *proofs* described in subsection 2.3.4.

The Isar - engine [Wen02] parses specification constructs and proofs and dispatches their treatment via the corresponding packages. Note that the Isar-Parser is configurable; therefore, the syntax for, say, a data-type statement and its translation into a sequence of logically safe constant definitions (constituting a "model" of the data type) can be modified and adapted, as well as the automated proofs that derive from them the characterizing properties of a data-type (distinctness and injectivity of the constructors, as well as induction principles) as `thm`'s available in the global context Θ thereafter. Specification constructs represent the heart of the methodology behind Isabelle: new specification elements were only introduced by "conservative" mechanisms, i. e. mechanisms that maintain the logical consistency of the theory by construction; internally these constructs introduce declarations and axioms of a particular form. Note that some of these specification constructions, for example type definitions, require proofs of methodological side-conditions (like the non-emptiness of the carrier set defining a new type).

We mention the last layer mostly for completeness: Recent Isabelle versions posses also an API written in Scala, which gives a general system interface in the JVM world and allows to hook-up Isabelle with other JVM-based tools or front-ends like the jEdit client. This API, called the "Prover IDE" or "PIDE" framework, provides an own infrastructure for controlling the concurrent tasks of proof checking. The jEdit-client of this framework is meanwhile customized as default editor of formal Isabelle *sessions*, i. e. the default user-interface the user has primarily access to. PIDE and its jEdit client manage collections of theory documents containing sequences of specification constructs, proofs, but also structured text, code, and machine-checked results of code-executions. It is natural to provide such theory documents as part of a certification evaluation documentation.

2.3.2 Isabelle and its Meta-Logic

The Isabelle kernel natively supports minimal higher-order logic called *Pure*. It supports for just one logical type prop the meta-logical primitives for implication $\_ \Longrightarrow \_$ and universal quantification $\bigwedge x.\ P\ x$. The meta-logical primitives can be seen as the constructors of *rules* for various logical systems that can be represented inside Isabelle; a conventional "rule" in a logical textbook:

$$\frac{A_1 \cdots A_m}{C} \tag{2.3}$$

can be directly represented via the built-in quantifiers $\bigwedge$ and the built-in implication $\Longrightarrow$ as follows in the Isabelle core logic Pure:

$$\bigwedge x_1 \ldots x_n .\ A_1 \Longrightarrow \ldots \Longrightarrow A_m \Longrightarrow \mathrm{C} \tag{2.4}$$

... where the variables x_1, ..., x_n are called *parameters*, the premises A_1, ..., A_m *assumptions* and C the conclusion; note that $\Longrightarrow$ binds to the right. Also more complex forms of rules as occurring in natural deduction style inference systems like:

$$\frac{\begin{array}{c}[A]\\ \vdots \\ B\end{array}}{A \to B} \tag{2.5}$$

can be represented by $(A \Longrightarrow B) \Longrightarrow A \to B$. Thus, the built-in logic provided by the Isabelle Kernel is essentially a language to describe (systems of) logical rules and provides primitives to instantiate, combine, and simplify them. Thus, Isabelle is a *generic* theorem prover. New object logics can be introduced by specifying their syntax and natural deduction inference rules. Among other logics, Isabelle supports first-order logic, Zermelo-Fraenkel set theory and the instance for Church's higher-order logic HOL. Moreover, Isabelle is also a generic system framework (roughly comparable with Eclipse) which offers editing, modeling, code-generation, document generation and of course theorem proving facilities; to the extent that some users use it just as programming environment for **sml!** or to write papers over checked mathematical content to generate LaTeX output. Many users know only the theorem proving language **isar!** for structured proofs and are more or less unaware that this is a particular configuration of the system, that can be easily extended. Note that for all of the aforementioned specification constructs and proofs there are specific syntactic representations in **isar!**.

Higher-order logic (HOL) [Chu40, And86, And02] is a classical logic based on a simple type system. It is represented as an instance in Pure. HOL

provides the usual logical connectives like $\_ \wedge \_$, $\_ \rightarrow \_$, $\neg \_$ as well as the object-logical quantifiers $\forall x.\, P\, x$ and $\exists x.\, P\, x$; in contrast to first-order logic, quantifiers my range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centred around extensional equality $\_ = \_ :: \alpha \Rightarrow \alpha \Rightarrow \text{bool}$. HOL is more expressive than first-order logic, since, e. g., induction schemes can be expressed inside the logic. Being based on a polymorphically typed λ-calculus, **hol!** can be viewed as a combination of a programming language like **sml!** or Haskel, and a specification language providing powerful logical quantifiers ranging over elementary and function types.

Isabelle/HOL is the session based on the embedding of HOL into Isabelle/Pure. Note The that simple-type system as conceived by Church for HOL has been extended by Hindley/Milner style polymorphism with type-classes similar to Haskel[WB89, Wen97].

2.3.3 The Isabelle Methodology and Specification Constructs

The core of the logic is done via an axiomatization of the core concepts like equality, implication, and the existence of an infinite set, the rest of the library is derived from this core by logically safe ("conservative") extension principles which are syntactically identifiable constructions in Isabelle files. In the following, we will briefly describe the most common conservative extension principles.

Conservative Extensions.

Besides the logic, the instance of Isabelle called Isabelle/HOL offers support for specification constructs mapped to conservative extensions schemes, i. e. a combination of type and constant declarations as well as (internal) axioms of a very particular form. We will briefly describe here *type abbreviations*, *type definitions*, *constant definitions*, *datatype definitions*, *primitive recursive definitions* as well-as *well-founded recursive definitions*. We consider this as the "methodologically safe" core of the Isabelle/HOL system.

Using solely these conservative definition principles, the entire Isabelle/HOL library is built which provides a *logically safe language base* providing a large collection of theories like sets, lists, Cartesian products $\alpha \times \beta$ and disjoint type sums $\alpha + \beta$, multi-sets, orderings, and various arithmetic theories which only contain rules derived from conservative definitions.

Type Abbreviations (Synonyms).

For example, typed sets are built in the Isabelle libraries via type synonyms on top of **hol!** as functions to bool; consequently, the constant definitions for

set comprehension and membership are as follows[2]:

```
type_synonym 'α set = 'α ⇒bool

definition Collect :: ('α ⇒bool) ⇒'α set
where    Collect S = S

definition member:: 'α ⇒'α set ⇒bool
where    member s S = S s
```

Isabelle's powerful syntax engine is instructed to accept the notation $\{x \mid P\}$ for Collect $\lambda x.\ P$ and the notation $s \in S$ for member $s\ S$. As can be inferred from the example, constant definitions are axioms that introduce a fresh constant symbol by some closed, non-recursive expressions; these types of axioms are logically safe since they work like an abbreviation. The syntactic side-conditions of the axioms are mechanically checked, of course. It is straightforward to express the usual operations on sets like $\_ \cup \_$, $\_ \cap \_ :: \alpha\,\text{set} \Rightarrow \alpha\,\text{set} \Rightarrow \alpha\,\text{set}$ as definitions, too, while the rules of typed set-theory are derived by proofs from them.

Datatypes.

Similarly, a logical compiler is invoked for the following statements introducing the types option and list:

```
datatype 'a list = Nil | Cons 'a 'a list
datatype 'a option = None | Some 'a
```

Here, [] and $a\#l$ are alternative syntax for Nil and Cons $a\ l$; moreover, $[a, b, c]$ is defined as alternative syntax for $a\#b\#c\#[]$. Similarly, the option type shown above is given a different notation: αoption is written as $\alpha_\perp$, None as $\perp$, and Some X as $\lfloor X \rfloor$. Internally, recursive datatype definitions are represented by type- and constant definitions. Besides the *constructors* None, Some, Nil and Cons, the statement above defines implicitly the match-operation case x of $\perp \Rightarrow F \mid \lfloor a \rfloor \Rightarrow G\ a$ respectively case x of $[] \Rightarrow F \mid (a\#r) \Rightarrow G\ a\ r$. From the internal definitions (not shown here) many properties are automatically derived like distinctness $[] \neq a\#t$, injectivity of the constructors or induction schemes.

Type definitions.

Type definitions allows for a safe introduction of a new type. Other specification constructs, for example datatype, are based on it. The underlying construction is simple: any non-empty subset of an existing type can be turned into new type. This is achieved by defining an isomorphism between

[2] To increase readability, the presentation is slightly simplified.

this set and the new type; the latter is introduced by two fresh constant symbols (representing the abstraction and the concretization function) and three internally generated axioms. As a simple example, consider the definition of type containing three elements. This type is represented by the first three natural numbers:

```
typedef three = {0::nat,1,2}
  apply (rule_tac x= 0 in exI)
  apply blast
  done
```

In order to enforce that the representing set on the right hand side is non empty, the package requires for this new type a proof of non-emptiness:

```
 typedef three = {0::nat,1,2}
  1. ∃x. x ∈{0, 1, 2}
```

To use this new type we need to finish the proof of non empty set started by the use of `typedef` which can be done differently. For example we can finish the proof using existing theorems on the logical operator ∃ in Isabelle/HOL. To see all Isabelle's theorems related to ∃ we use the Isabelle command `find_theorems`. The query searches for theorems whose name contains an "ex" sub-string. One of the results is:

```
 find_theorems name : exI
 HOL.exI: ?P ?x ⟹∃x. ?P x
```

The searched theorems is applied in the following. In our case, the Isabelle proof method *rule_tac* is used, a resolution step, which unifies the theorem HOL.exI against the first proof goal in a resolution step:

```
 apply (rule exI[where x= 0])
 apply blast
 done
```

Its application in the proof allows to replace the schematic variable $?x$ by the constant 0 in our proof; this is specified by the key word **in** followed by the name of the theorem. The other schematic variable $?P$ is automatically filled in (using higher-order unification), which is possible since only one solution remains. The remainder of the proof consists of a call to the highly automated method `blast`, which does the trick for the necessary set-theoretic proof.

It remains to point out that the same proof can be done by different proof-style called *structured proof* or *Isar-proof*. The same proof can be represented in this style as follows:

```
typedef three = {0::nat,1,2}
  proof
  show 1 ∈{0, 1, 2}
    by blast
  qed
```

After finishing the proof about the definition of this new type, many theorems will be deduced automatically by Isabelle. We can check the new deduced theorems related to this new type by using the command `find_theorems`. In the concrete example, there are 82 new theorems deduced that were related to this type definition.

```
find_theorems name : three.
searched for name: three
found 9 theorems (40 displayed)
```

Well-founded Recursive Function Definitions.

Actually, there is a parser for primitive and well-founded recursive function definition syntax. For example, the sort-operation can be defined by:

```
fun ins ::
    'α:: linorder ⇒'α List.list ⇒'α List.list
where
  ins x [] = [x]
| ins x (y#ys) = (if x < y then x #y#(ins x ys) else y#(ins x ys))

fun sort ::'α:: linorder List.list ⇒'α List.list
where
  sort  [] = []
| sort (x#xs) = ins x (sort xs)
```

which is again compiled internally to constant definitions. Here, $\alpha :: \text{linorder}$ requires that the type α is a member of the *type class* linorder. Thus, the operation sort works on arbitrary lists of type $(\alpha :: \text{linorder})$ list on which a linear ordering is defined. The internal (non-recursive) constant definition for the operations ins and sort is quite involved and requires a termination proof with respect to a well-founded ordering constructed by a heuristic. Nevertheless, the logical compiler will finally derive all the equations in the statements above from this definition and makes them available for automated simplification.

The theory of partial functions is of particular practical importance. Partial functions $\alpha \rightharpoonup \beta$ are then defined as functions $\alpha \Rightarrow \beta$ option supporting the usual concepts of domain dom $f \equiv \{x \mid f\ x \neq \text{None}\}$) and range ran $f \equiv \{x \mid \exists y.\ f\ y = \text{Some}\ x\}$. Partial functions can be viewed as "maps" or dictionaries; the *empty* map is defined by $\emptyset \equiv \lambda x.$ None, and the update operation, written $p(x \mapsto t)$, by $\lambda y.\, \text{if}\ y = x\ \text{then Some}\ t\ \text{else}\ p\ y$. Finally, the override operation on maps, written $p_1 \oplus p_2$, is defined by $\lambda x.\, \text{case}\ \ p_1\ x\ \ \text{of None} \Rightarrow p_2 x \mid \text{Some}\ X \Rightarrow \text{Some}\ X$.

Records

An Isabelle record [Wen15, NWS+] is a data structure that contain a number of fields. In its essence a record are tuples, where the fields are selectors in its components. Internally, Isabelle generates for this specification construct a theory describing records selectors, records update functions, a more field and a records refinement scheme. The Isar syntax for declaring records is:

```
record ('a, 'b) state =
  field1   :: 'a
  field2   :: 'b
```

In this example we had declared an Isabelle record type named *state*, that contain two fields *field1, field2* and supports two types *'a, 'b*. After the definition of this record type a set of Isabelle theorems are generated automatically and added to Isabelle gobal context.

```
Record1.state.select_defs(1):
  field1 ≡
     id ∘ Record.iso_tuple_fst Record.tuple_iso_tuple ∘
     Record.iso_tuple_fst state_extᵢuple_Iso
```

This theorem is used to retrieve field named `field1` from the record `state`. Another theorem used to update the same field is generated automatically, example:

```
Record1.state.update_defs(1):
 field1_update ≡
     Record.iso_tuple_fst_update state_extᵢuple_Iso ∘
    (Record.iso_tuple_fst_update Record.tuple_iso_tuple ∘id)
```

Other operations on records like extending record type are defined too.

Function Definitions

The HOL instantiation for Isabelle contains a theory on total functions [Nip12]. A set of operations and lemmas are defined in this theory. An Isabelle function is seen as an application $f\colon E \to F$, where E is the domain

and F is the range of f, in the following some Isabelle definition that exist in this theory are presented:

```
definition id :: 'a ⇒'a where
  id = (λx. x)
definition comp :: ('b ⇒'c) ⇒('a ⇒'b) ⇒
                   'a ⇒ 'c (infixl o 55)
where f o g = (λx. f (g x))

lemma id_apply [simp]: id x = x
  by (simp add: id_def)

lemma comp_apply [simp]: (f o g) x = f (g x)
  by (simp add: comp_def)
```

In those two examples `id` specify the identity function and`comp` (has as infix syntax the symbol `o`) specify function composition. Actually, the theory *Fun.thy* is an extension of the *Set.thy* theory, other definitions like domain of the function, `range`, `image` ... are implemented in *Set.thy*.

ML Code.

It is possible inside Isabelle documents to directly access the underlying ML-layer of the system architecture, and even extend the environment of the underlying ML interpreter/compiler. One can include the fragment:

```
ML{* fun fac x = if x = 0 then 1 else x * fac(x-1); *}
```

in a document and then later on evaluate:

```
ML{* fac 20; *}
```

Since Isabelle itself sits as a collection of ML modules in this SML environment, it is possible to access its kernel and tactical functions:

```
ML{* open Tactic;
     fun mis x = res_inst_tac [(x, x)] {@thm exI} 1*}
```

which defines a new tactic that applies just the existential-introduction rule of **hol!**. This is the key to build large and own tactic procedures and even tools inside the Isabelle environment. Note that the fragment `{@thm exI}` is called an *antiquotation*; it is expanded before being passed to the SML compiler with code that accesses the `thm exI` (see section subsection 2.3.3, pp8.) in the Isabelle database for theorems. By additional SML-code, this tactic can be converted into a Isar-*method*, which can be bound to own syntax inside the Isar-language. Thus, the proof language is technically extensible by own, user-defined proof-commands (see [Wen15] for the details).

2.3.4 Isabelle Proofs

In addition to types, classes and constants definitions, Isabelle theories can be extended by proving new lemmas and theorems. These lemmas and theorems are derived from other existing theorems in the context of the current theory. Isabelle offers various ways to construct proofs for new theorems, we distinguish two main categories: forward and backward proofs:

Local forward proofs.

The goal of a forward proof is to derive a new theorem from old ones. This is done either by instantiating some unknowns in the old theorems, or by composing different theorems together.

The instantiation can be done using the `of` and `where` operators as follows: `thm[of inst1 inst2 ...]` or `thm[where var1=inst1 and var2=inst2 ...]`. If we consider for example the existential introduction theorem called `exI` and given by `?P ?x` $\Longrightarrow$`∃ x. ?P x`. The unknown variable `x` can be instantiated with a fixed variable `a` using the following command `exI[of _ a]` which is equivalent to `exI[where x=a]`. Note that when using `of` the instances of the variables appear in the same order of appearance of the unknown variables in the theorems. Consequently, we can avoid instantiating a variable by giving a dummy value in the position of its corresponding instance.

The second way of deriving theorems is by composing different theorems together using the `OF` or `THEN` operators. The first operator `OF` is used to compose one theorem to others. For a theorem `th1` given by `A` $\Longrightarrow$`B` and a theorem `th2`given by `A'`, the theorem `th1[OF th2]` results from the unification of `A` and `A'` and thus instantiating the unknowns in `B`. Theorems with multiple premises can be composed to more than one theorem given as arguments to the `OF` operator. For example, given the conjunction introduction theorem `conjI` given by `?P` $\Longrightarrow$`?Q` $\Longrightarrow$`?P` $\wedge$ `?Q` and the reflexivity theorem `ref` given by `?x = ?x`, the composition of these theorem `conjI[OF refl[of a] refl[of b]]` results in the following theorem `a = a` $\wedge$ `b = b`. In a similar way, the `THEN` operator is used to compose different theorems together. The theorem `th1[THEN th2]` is obtained by applying the rule `th2` to the theorem `th1`. For example, composing a theorem `th1` given by `a = b` with the symmetry rule `sym` given by `?s = ?t` $\Longrightarrow$`?t = ?s` is written `th1[THEN sym]` and the result is `b = a`.

Global backward proofs.

The usual and mostly used proof style is the backward or goal-directed proof style. First, a proof goal is introduced then the proof is performed by simplifying this goal into different sub-goals and, finally, prove the resulting sub-goals from existing theorems. The proofs are build using natural de-

duction by applying some existing (proved) inference rules. For each logical operator, two kinds of rules are defined: introduction and elimination rules. The backward proofs can be structured in two different ways:

1. Apply style proofs, where the proof goal is simplified using a succession of rules applications. This results in a so-called apply-script, describing the proof steps. An example of such a proof is given in the following:

```
lemma conj_rule: ⟦P; Q ⟧⟹ P∧ (Q ∧P)
  apply (rule conjI)
    apply assumption
  apply (rule conjI)
    apply assumption
    apply assumption
  done
```

   Although this proof style is easy to apply, long apply-scripts can become unreadable and hard to maintain. A more structured and safe way to write the proofs is by using the Isar language.

2. Structured Isar proofs allow for writing sophisticated and yet still fairly human-readable proofs. The Isar language defines a set of commands and shortcuts that offer more control on the proof state. An example of a structured induction proof is given in the following:

```
lemma
  fixes n::nat
  shows 2 * (∑ i=0..n. i) = n * (n + 1)
  Proof (induct n)
    case 0
      have 2 * (∑ i=0..n. i) = (0::nat)
        by  simp
      also have (0::nat) = 0 * (0 + 1)
        by simp
      finally show ?case .
  next
   case (Suc n)
     have  2 * (∑ i=0..Suc n. i) = 2*(∑ i=0..n. i) + 2 *(n + 1)
       by simp
     also have 2*(∑ i=0..n. i) = n * (n + 1)
       by (rule Suc.hyps)
     also have n * (n + 1) + 2 * (n + 1) = Suc n * (Suc n + 1)
       by simp
     finally show ?case .
  qed
```

For the sake of this presentation, we appeal to an "immediate intuition" of a mathematically knowledgable reader; for detailed introduction into the structured proof language, the reader is referred to the Isar Reference Manual of the System documentation.

In addition to internal Isabelle proof procedures, there are some external proof procedures (`blast` going back to leantap[BP95], `metis` existing as a stand-alone first-order paramodulation procedure[Hur03] as well as CVC4[BCD+11] and Z3[dMB08] via the smt interface) that have been integrated into Isabelle in a logically safe way.

2.3.5 Isabelle/HOL Code Generation

Finally, Isabelle/HOL manages a set of *executable types and operators*, i. e., types and operators for which a compilation to **sml!**, OCaml, Scala, or Haskel is possible. Setups for arithmetic types such as int have been done allowing for different trade-offs between trust and efficiency. Moreover any datatype and any recursive function are included in this executable set (providing that they only consist of executable operators). Of particular interest for evaluators is the use of the Isar command:

$$\textbf{value} \quad \text{sort}[1,7,3] \tag{2.6}$$

In the context of the definitions, it will compile them via the code-generator to SML code, execute it, and output:

$$[1,3,7] \tag{2.7}$$

This provides an easy means to inspect constructive definitions and to get easy feedback for given test examples for them. See the part "Code generation from Isabelle/HOL theories" by Florian Haftmann from the Isabelle system documentation for further details.

2.3.6 Isabelle/HOL Document Generation

Of particular interest for evaluators or certifications are Isabelle's features for semantically supported typesetting: within the document element:

```
1    text{* This is text containing $\lambda$'s and $\beta$'s ... *}
```

for example, arbitrary LaTeX code can be inserted for using technical and mathematical notation of annotations of formal document elements. Inside a text-document, the *document antiquotation* mechanism already mentioned in 2.3.3 can be applied:

```
1    text{* Text containing theorems like {@thm exI} ... *}
```

which results in a print of theorems directly from their formal Isabelle presentation. Since it is possible to define new antiquotations, one can, for example, track security requirements or security claims in theorems or tests (a detailed description of document antiquotations is found in the "Isar Reference Manual" by Makarius Wenzel from the Isabelle system documentation). Thus it is possible to use this mechanism to support the traceability of the common criteria items like protection profiles, security targets, requirements, security properties etc. For all these entities, be it informal or not, declarations and applications of antiquotations can be used in text fragments that allow for a direct consistency checking over the entire document.

During a certification process, evaluators are encouraged to use the Isabelle/jedit user-interface directly (and not just the generated .pdf documentation), since it allows for an in-depth inspection and exploration of the formal content of a theory: tooltips reveal typing information, evaluations of critical expressions can often be done by the `value ...` document item, and operator-symbols occurring in HOL-expressions were hyper-linked to referring definitions or binding occurrences. Note, however, that a user-interface is a dozen system layers away from a Isabelle inference kernel which opens the way for implementation errors in display and editing components, increasing the risk of misinterpretations. A final check of an entire document should therefore be made in the (GUI-less) build mode (which enforces also stronger checking).

2.3.7 Isabelle extensions: HOL-TestGen

HOL-TestGen[3](see Figure 2.4) is an interactive, i. e., semi-automated, test generation tool for specification-based tests built upon Isabelle/HOL. Instead of using Isabelle/HOL as "proof assistant," it is used as modeling environment for the domain specific background theory of a test (the *test theory*), for stating and logically transforming test goals (the *test specifications*), as-well as for the test generation method implemented by Isabelle's tactic procedures. In a nutshell, the test generation method consists of:

1. a *test case generation* phase, which is essentially an equivalence partitioning procedure of the input/output relation based on a **cnf!**-like normal form computation,

2. a *test data selection* phase, which essentially uses a combination of constraint solvers using random test generation and the integrated SMT-solver Z3 [dMB08],

3. a *test execution* phase, which reuses the Isabelle/HOL code-generators

[3]HOL-TestGen was never used to: test complex real systems, and concurrent code before this thesis

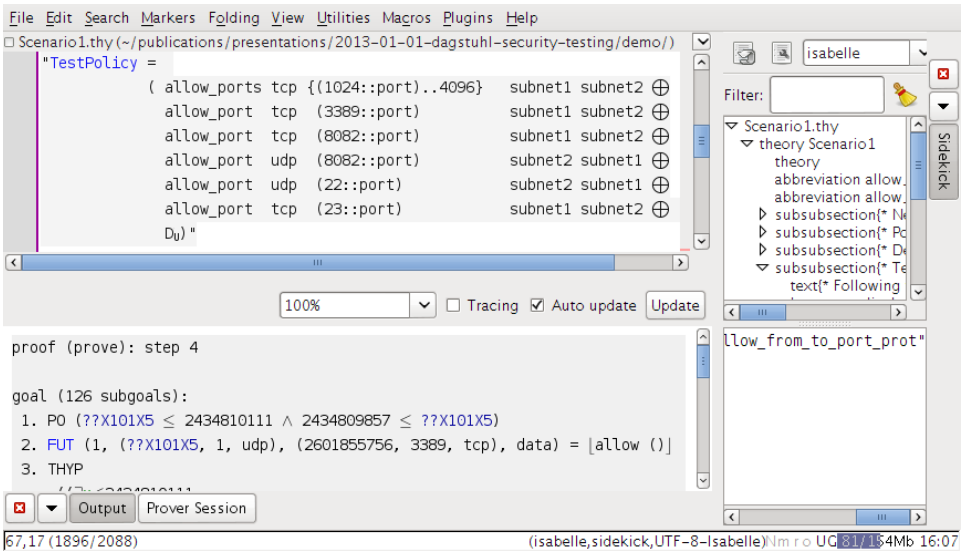


Figure 2.4: An Isabelle session showing the jEdit client as Isabelle Interface. The upper-left sub-window allows one to interactively step through a test theory comprising test specifications while the lower-left sub-window shows the corresponding system state of the spot marked in blue in the upper window.

> to convert the instantiated test cases to test driver code that is run against a system under test.

A detailed account on the symbolic computation performed by the test case generation and test selection procedures is contained in [BW13]. The test case generation method is basically an *equivalence partitioning* combined with a variable splitting technique that can be seen as an *(abstract) syntax testing* in the sense of the ISO 29199 specification [Int12, Sec. 5.2.1 and 5.2.4].

The equivalence partitioning separates the input/output relation of a program under test (PUT), usually specified by pre- and post-conditions, into classes for which the tester has reasons to believe that PUT will treat them the same.

Of course, the HOL-TestGen approach inherits all glory, but also all limitations of a testing approach: The entire specification is reduced via specific *test purposes* and underlying *test hypothesis* ("pick one out of the equivalence class, and it's going to be ok for all class members") to a *finite* number of tests to be checked. These purposes and hypotheses ' may be difficult to justify and need careful inspection, more difficult than having just a universal statement over the entire input/output relation. On the other hand, testing can establish confidence over the **real system**, and makes no modeling

assumptions — like the Simpl-approach subsection 3.4.1 — over the underlying hardware, the correct modeling of behavior of hardware components such as sensors, the compiler, and the equivalence of the assumed operational semantics of the used programming language(s) with the actually executed one. For this reason, it can be safely stated that for certifications of the highest-levels, a suitable *combination* of test and proof techniques will be necessary. Proofs for the higher levels of the models establishing the desired security properties in a *Target Of Evaluation* TOE, tests for establishing that the assumptions made in the lower levels of the models correspond to the reality in the TOE.

2.4 The Verified Architecture Microprocessor (VAMP)

The Verified Architecture Microprocessor (VAMP) as well as the micro-kernel VAMOS [Dor10] has been developed and verified in the context of the German research projects Verisoft[4] and VerisoftXT[5]. The goal in particular of the former project was the pervasive formal verification of computer systems from the application level down to the silicon, i. e., the hardware design.

On the *Application Software Layer*, this includes foundational proofs justifying a verification approach for system-level concurrent programs that are running as user processes on the micro-kernel VAMOS [Dor10]. On the *System Software Layer*, VAMOS provides an infrastructure for memory virtualization, for communication with hardware devices, for process (represented as a sequence of assembly instructions), and for inter-process communication (IPC) via synchronous message passing that need to be verified. On the *Tools Layer*, the correctness of the compiler needs to be verified and, finally, on the *Hardware Layer*, the functional correctness of the hardware design is formally verified.

These four layers comprise the Verisoft Architecture (see Figure 2.5); each of the layers is in itself structured in several sub-layers.

Our work focuses on the hardware layer, more precisely the assembly-level (VAMPasm), i. e., the instruction set of the Verified Architecture Micro-Processor (VAMP) [BJK+06]. VAMP is a pipelined reduced instruction set (RISC) processor based on the out-of-order execution principle (see [HP06] for details). The VAMPasm (section 5.2 presents the formal model we are using in our work) includes 56 instructions: 8 instructions for memory data transfer, 2 instructions for constant data transfer, 2 instructions for register data transfer, 14 instructions for arithmetic and logical operations, 16 instructions for test operations, 6 instructions for shift operations, 6 instruc-

[4] www.verisoft.de
[5] www.verisoftxt.de

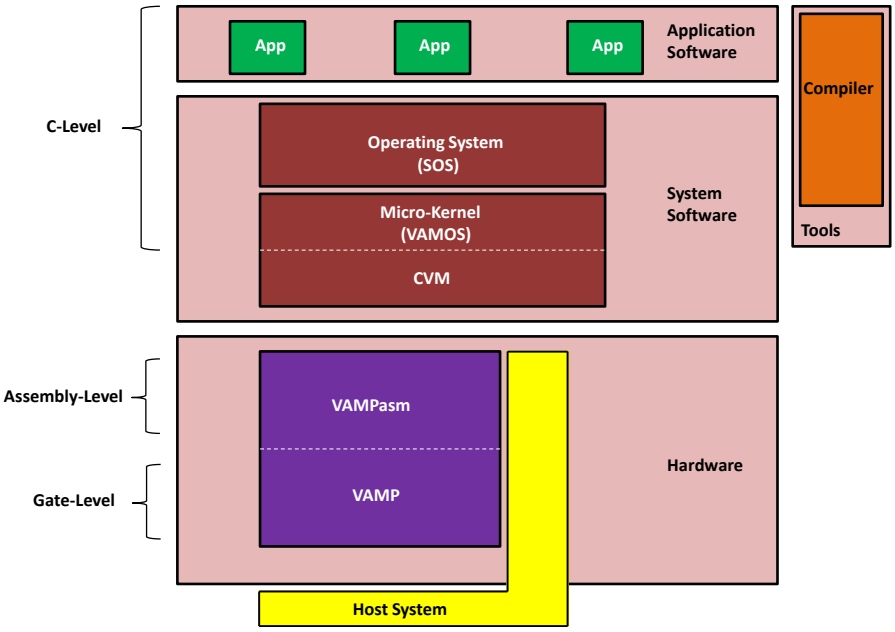


Figure 2.5: The Verisoft System Layers.

tions for control operations as well as 2 instructions for interrupt handling. In our unit and sequence test scenarios presented in section 5.3, we generate tests from a formal model of the instruction set, i. e., we test the conformance of the gate level (which corespondents to the implementation in traditional model-based testing) to assembly-level (which corespondents to the model in traditional model-based testing).

2.5 PikeOS System Architecture

PikeOS is a real-time commercial operating system that supervises and ensures the execution and separation between software applications running on the top of various hardware platforms [SYS13a, SYS13b]. It stands in the tradition of so-called *separation kernels* and follows ideas of the influential L4 kernel project [Lie95]. The PikeOS architecture comprises four layers (see Figure 2.6). The *virtual machine initialization table* (VMIT) is a database containing the global configuration of the system and its application structure. In the VMIT, *partitions* (virtual machines), *tasks* (POSIX-like processes), their *threads*, their memory-, processor-, and time resources, communication channels as well as access-control rights on these resources were defined. Only at boot-time, partitions, processes and threads can be created via *PikeOS System Software* (PSSW); at run-time the application structure and its time-scheduling is fixed: PikeOS has no dynamic process creation. In

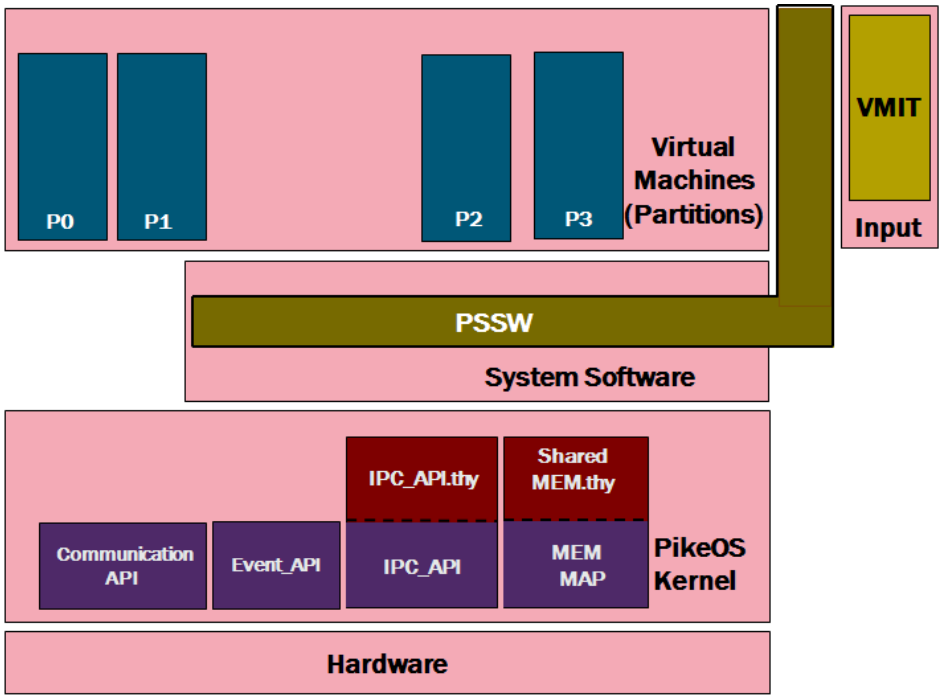


Figure 2.6: PikeOS architecture.

other words: based on the VMIT configuration, the *PikeOS system software* (PSSW) will generate a set of virtual machines in the *Partitions layer* during the boot-phase. In this layer each resource partition is composed from a set of applications, and can be executed under the predefined policy and use the predefined resources of the VMIT. Applications in the resource partitions can also be used for system calls of *PikeOS kernel*. In kernel layer, the set of resource partitions is seen as a set of PikeOS *tasks*, that contain PikeOS *threads* and shares kernel resources (memory, files, processors, communication channels . . .).

The kernel provides a set of APIs used by the threads and tasks. As in Unix-like systems, special hardware—the MMU—gives application-level *tasks* the illusion to live in an own separate memory space: the virtual memory. However, all *threads* belonging to a task live in the same memory space, namely the memory space of the task they belong to. In contrast, system-level tasks can also access the physical memory and the MMU. Besides memory separation, PikeOS also offers time-separation and multi-core support.

Our work focuses on a particular part of the kernel layer providing inter-process communication (IPC), the PikeOS IPC API.

2.6 Conclusions

In this chapter we have presented the general context of this thesis. First, an overview on formal testing, its background and its relation with the other formal methods techniques were introduced. Actually, formal testing approaches bridge the gap between the formal model and the reality; in certification effort, this can be a valuable contribution to gain confidence over results achieved by deductive verification or model checking, in which, the verification activity is usually done on the model level solely. From the other side, deductive verification and model checking approaches, can be used to refine/adapt an additional model to a testable one, i. e. a model that is suited to symbolic execution and that can therefore be used in the different testing approaches.

Second, we have presented Isabelle/**hol!** and pointed the essential system features. We believe that a such introduction to Isabelle system will help the reader to understand our contributions explained in the remaining chapters. Finally, the basic notions and the system architecture of VAMP Processor and PikeOS system were described.

The next chapter contain our first contribution during this thesis, in particular we will introduce an instantiation of the text in [JH08] for Isabelle in order to show how Isabelle can be used in certification processes.

Part II

Contributions

3

A sideline : Isabelle/HOL in certification processes A System Description and Mandatory Recommendations

Contents

3.1 Introduction

Recently, theorem proving environments have been widely used in the area of computer systems security and certification and, for instance, in Common Criteria. The Common Criteria (CC) [Mem06] is a well-known and recognized computer security certification standard. The standard is centered around the role of the *developer*, who provides implementation but also "artefacts of compliance with the level of security targeted", while the *evaluator* "confirms the compliance of the information supplied" as well as determines "completeness, accuracy and quality" of the deliverables.

Especially wrt. "completeness, accuracy and quality" of specifications and proofs, formal methods and especially mechanically proof checking techniques can push the trust and the reproducibility of the results to levels not obtainable by a human certification expert alone. This explains why at its higher assurance levels, the CC requires the use of formal methods for specification and verification. A well-established formal specification formalism must be used to model the system and of the different security policies. A reliable theorem prover is needed to prove and verify different properties of the specification. Recent theorem provers offer rich and powerful formal environments that are very suitable for both activities.

Among the important number of theorem provers available nowadays, we concentrate on the Isabelle theorem prover[1]. Following [Hal08], the Isabelle System, developed into one of the top five systems for the logically consistent development of formal theories. In particular the instance of the Isabelle system with higher-order logic called Isabelle/HOL is therefore a natural choice as a formal methods tool as required by the Common Criteria on the higher assurance levels EAL5 to EAL7.

In this chapter we present a side-effect of our work that still relevant to its context (a European project aiming at a certification of an industrial operating system), as well as the methodological role of testing a certification process. In particular we contributed to the paper[2] which was sent to the ANSSI[3]. As a contribution, the chapter culminates in some high-level mandatory guidelines and recommendations for both developers and evaluators of certification documents using Isabelle. It attempts to be a complement to [JH08].

[1] At time writing, the current version is Isabelle2013-2.

[2] http://www.euromils.eu/downloads/Deliverables/Y2/2015-EM-UsedFormalMethods-WhitePaper-October2015.pdf

[3] http://www.ssi.gouv.fr/

The chapter proceeds as follows: at first in section 3.2, we give some general information from Common Criteria standard about formal methods, modeling and associated requirements. In section 3.3, we refer to methodological issues of Isabelle/HOL leading to recommendations for evaluators. In section 3.4 we chose a major extensions of Isabelle for code-verification, and discuss its advantages and limits in a high-level certification process. The final discussion contains a little survey on publications on the topic as well as a summary for evaluators.

3.2 Common Criteria: Normative Context

For high levels of certification (i.e. for EAL5 to EAL7) in the Common Criteria [Mem06] some requirements introduce the use of formal methods at diverse phases of the design process. Regarding to the level of security target required, the use of formal methods match different objectives. For deeper explanations on high certification levels related to Common Criteria, i. e. EAL 5 to EAL7, and their requirements we would refer to [YABC15].

3.3 Methodological Recommendations for the Evaluator

There are four potential dangers of a formal proof system that it wrongly accepts the desired theorem "This operating system is secure":

1. Inherent inconsistency of the logics (e. g., **hol!**) or inconsistent use of the logics (introduction of inconsistent axioms by one way or the other).
2. The incorrect implementation of Isabelle the Isabelle Kernel and of the **hol!** instance in it.
3. The incorrect package implementation realizing advanced specification constructions like type definitions etc.
4. Since Isabelle is highly configurable, there is a certain danger of obfuscation of bogus-proofs.

Beyond the more philosophical objections[4], the risk outlined by the by first item is in fact **minimal**: Higher-order logic is an extremely well studied object of academic interest [And86, GM93], and while there are known limits in proving soundness and completeness inside a **hol!**-prover, they just stimulated a lot of recent research to come a "formal proof over **hol!** in **hol!**" as

[4]For example, the fundamental doubt in the existence of infinite sets[And86]...

close as possible, e.g. by adding to **hol!** an axiom over the existence of a sufficiently large cardinal [Har06, MOK13].

The risk outlined by the second item is also **very small**. The reasons are threefold:

A Some of the aforementioned soundness proofs cover also the implementation aspects of the core of a provers of the **hol!**-family (**hol!**-light, ...).

B The specific architecture of provers of the LCF family (HOL4, Isabelle, HOL-light, Coq) enforces that any proof is actually checked by by this fairly small core.

C These core-inferences can optionally be protocoled in an proof-object which can, in principle, in case of serious doubt be checked by another implementation of a **hol!**-prover. However, since these objects tend to be very large, this approach requires decent engineering. Fortunately, this should only be necessary in exceptional cases.

The risk of the third item is **minimal** as far as the described standard conservative standard extension schemes such as `type_synonym`'s, `datatype`'s, `definition`'s and `fun`'s, `typedef`'s, `specification`'s, `inductive`'s, typeclasses and locales are concerned. The same holds for diagnostic commands like `type`, `term`, `value`, etc. that do not change the global context of a theory. These are fairly well-understood schemes which have in parts been proven formally correct for similar systems such as the HOL4 system[KAMO14]. These schemes cover the largest parts of the Isabelle/HOL libraries. Here lies the main advantage of the LCF-approach and the methodology to base libraries on conservative (logically safe) definitions.

The risk is **small** as far as other standard extension schemes are concerned; since extension schemes generate internally axioms, there have been reported consistency problems with combinations of other extension schemes such as `consts` and `defs` as well as `defs (overloaded)`; the Isabelle reference manual points out that the internal checks of Isabelle do not guarantee soundness.[5]

It remains the risk of item four, which is concerned with the resulting methodology in "how to use Isabelle". For very large theory documentations, it must be considered **non-negligeable**. It is the key-issue addressed in the remainder of this section.

3.3.1 On the use of SML

As mentioned earlier, Isabelle is an open environment that allows via

[5] See Isabelle Isar-Reference Manual (Version 2013-2, pp. 103): "It is at the discretion of the user to avoid malformed theory specifications!"

```
ML{* SML ML code *}
```

to include arbitrary SML programs, in particular programs that make direct inferences on top of the kernel. This use of Isabelle is not unsafe; critical parts of the **hol!** library use this mechanism. Isabelle is designed to have user land SML code extensions, and the kernel protects itself against logical inconsistencies coming from ML extensions. However, there are a few deliberate opt-outs, and furthermore, it is in principle possible to obfuscate them in Isabelle ML code such that an evaluator may be fooled by a text appearing to be an Isabelle proof but isn't in the sense of the inference kernel. Thus, besides the principle possibility that a pretty-printed theorem does not state what it appears to state by some misuse of mathematical notation (an inherent problem of any formal method), there is the possibility of fake-proofs as a consequence of ML code and (re)-configurations of the ISAR proof language.

If SML-code is accepted in an evaluation, it has to be made sure — potentially by extra justifications or external experts with Isabelle implementation expertise — that this code does not implicitly generate axioms, registers oracles and defines proof methods equivalent to `sorry` (or variants like `sorry_fun`) to be discussed in the sequel; in any case, the evaluation is substantially simpler if SML-code is strictly avoided.

3.3.2 Axioms and Bogus-Proofs

Obviously, when using the Isar `axiomatization` construct allowing to add an arbitrary axiom, it is immediately possible to bring the system in an inconsistent state. The immediate methodological consequence is to ban it from use in to be evaluated theories completely (such that it is only internally used inside specification constructs in and and in the aforementioned foundational axioms coming with the system distribution) and to restrict theory building on conservative extensions. This is also common practice in scientific conferences addressing formal proof such as ITP.

However, there are more subtle ways to introduce an axiom that leads to inconsistency. First, there is a mechanism in Isabelle to register *oracles* into the system. They can be used for a particularly simple, but logically unsafe integration of external provers into Isabelle and can be used inside self-defined tactics. Logically, an oracle is a function that produces axioms on the fly. It is an instance of the axiom rule of the kernel, but there is an operational difference: The system always records oracle invocations within proof-objects of theorems by a unique tag. Of course, oracle invocations should again be avoided in a certified proof.

A particular instance of the oracle mechanism is the `sorry` proof method. This is method is always applicable and closes any (sub)-proof successfully, and a useful means in top-down proof developments in Isabelle. Unneces-

sary to repeat that no sorry statements should remain in a proof document underlying certification. By the way, the system is by default in a mode in which it refuses to generate proof documents containing sorry's, only by explicitly putting it in a mode called quick_and_dirty this can be overcome. There are several ways to activate quick_and_dirty, by it by explicit ML statements like quick_and_dirty:=true, be it in the ROOT.ML-files (till version 2013-1), or be it in the session- configuration files ROOT-files (since version 2013).

Oracles and sorry's are particularly dangerous in methodological foundation proofs (type or type-class is non-empty, recursions well-founded), since the use of the the oracle-tag inside the corresponding proof-objects gets lost on the level of type expressions. Thus, a sorry could introduce inconsistent types whose "effects" could be used in bogus-proofs depending on them.

We will discuss this a little more in detail: Recall that deduction in Isabelle/**hol!** is centered around the requirement that types and type-classes are non-empty. This is a consequence of the fact that the β-reduction rule $((\lambda\, x :: \tau.E)E' \rightarrow E[x := E'])$ is executed pervasively during deduction, be in in resolution or rewriting steps. It is well-known however, that β-reduction is unsound in the presence of empty types[6]. Thus, an obfuscated sorry in a methodological proof leaves no other than very local traces in the proof objects and can be exploited much later via an inconsistent type in a proof based on this type definition; the exploit could again be obfuscated by another self-defined proof-method, say auto' which will be hard to detect by inspection. The only systematic way to rule out obfuscated bogus-proof is either by ruling out ML-constructs or by checking *all* proof objects of the entire theory.

3.3.3 On the use of external provers

The Isabelle distribution comes with a number of external provers, namely:

- sledgehammer : its use is uncritical, since it remains completely extern to proof documentations and is only used for the generation of high-level Isabelle proofs, that were certified by the kernel.
- blast, metis: these are internal devices but also uncritical, since their results were used via a proof object certification.
- smt: this method uses, for example, the external SMT-solver Z3. The integration is carefully made and uses no oracles - instead, a form

[6]Consider the case of τ having a semantic interpretation into an empty set $I(\tau) = :$ then the semantic interpretation of the function $(\lambda\, x :: \tau.E)$ must be in the function space: $^D =$ where D is the space of interpretations for the type τ' of E. Obviously, there is no possible result for the application ...

of tactical proof re-construction mechanism is used [BW10] that is logically safe.

Other external provers have to be considered carefully; in particular integrations using the oracle-mechanism should be ruled out.

3.4 Extensions of Isabelle: Guidelines for the Evaluator

Besides HOL-TestGen described in subsection 2.3.7, there are other Isabelle extensions relevent for certification processes, namely Isabelle/simpl.

3.4.1 An Example: The Isabelle/Simpl

Isabelle/Simpl is an verification environment built conservatively on Isabelle/HOL. It supports a sequential imperative programming language, for which it defines its syntax, semantics, Hoare Logics and a verification condition generator (again derived), which form together a complete verification environment. Together with an (untrusted) parser that compiles C programs into Isabelle/Simpl[GAK12], this particular environment follows a similar program verification technique like Frama-C/Why/AltErgo ([CKK+12, FP13],alt-ergo.lri.fr) or VCC/Boogie/Z3[BW10].

The entire environment is part of the Isabelle-oriented "Archive of formal Proofs", see afp.sourceforge.net in general and afp.sourceforge.net/entries/Simpl.shtml in particular.

The environment has been used for one of the most ambitious code-verification projects recently, the verification of the L4-Microkernel (cf. www.ertos.nicta.com.au/research/l4.verified, [KEH+09]).

In itself, Isabelle/Simpl can be considered nearly as as "trustable" as Isabelle/HOL itself : the library is built upon conservative extensions of the HOL -kernel, and the ML extensions are done by Isabelle developers themselves and stood the test of the time. Program verification proofs establishing that a Simpl-program is correct with respect its (pre-post-condition) specifications can be handled by the same evaluation procedures as any other Isabelle development.

However, as in any process involving the verification of C programs, the C parser and its transition from "real C" to the idealized imperative language Simpl has to be considered with a wise dose of scepticism. Here is are whole spectrum of different glimpses possible: since the C parser defines a semantics-by-translation for its fragment of C, the question remains unproven that this semantics is faithful to the semantics of the real C compiler generating production-level code (which involves questions on compiler cor-

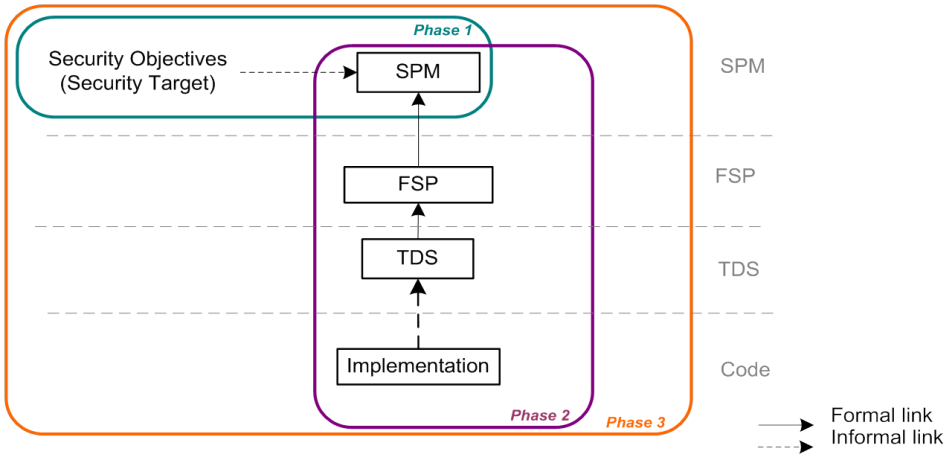


Figure 3.1: Refinement steps for a formal development approach compliant to CC

rectness, semantic faithfulness of the execution environment, correctness of compilation optimizations, hardware-correctness, etc.). The problem has been addressed via particular validation techniques of the parsing process [GAK12], but is, in full generality, unsolvable.

3.5 Recommendations for CC certifications

3.5.1 A refinement based approach for CC evaluation

The figure 3.1 presents a refinement scheme which implements different refinement steps from security policy model SPM to implementation. With this approach, the properties demonstrated on an abstract SPM are formally preserved down to the levels of the functional specification model FSP and a TOE specification design model, the TSD. At each level of abstraction the dedicated model and its associated proofs demonstrate the security properties and are compliant with the CC requirement. The use of a formal refinement methodology demonstrates the consistency between each refined model and preserve the properties demonstrated at high level of abstraction.

The evaluation of this kind of approach can be conducted in three different phases by the evaluator:

- Phase 1: Verification of the proof of the SPM formal specification. On the initial abstract model, a verification shall be conducted to check the relevance of the security objectives modeling in the formal model with the informal specification. A second point is the verification of the model soundness to assure than the model is not inconsistent (refers

to chapter 3.3.2).

- Phase 2: Refinement of the SPM formal specification. A first step of this phase is the verification of the refinement process and methodology. On each refinement, verification on the properties and on the soundness of the model are conducted. From the initial abstract model, on each intermediate concrete model, the evaluator checks the traceability (i.e. the traceability of the requirements) between models. An informal link can be considered between the last formal model of the TDS and the implementation. A bi-directional detailed traceability of the security requirements shall be managed between this two different artefacts to verify the implementation of the security requirements and than the implementation contains only desired requirements [7].

- Phase 3: General and transverse activities. This last phase consists mainly of the verification on the proofs and on justifications on the tools used as support for development and design. The complete traceability from the security target to the implementation is verified included traceability between each refinement steps of formal models. During this phase, the evaluator replay the proofs and check the consistency of the formal properties and assumptions defined on the environment and the context (see 2.3.5 last paragraph for details of facilities supplied by Isabelle/HOL. The use of keywords to report the proof of parts of the proof obligations is forbidden (for example the use of the `sorry` proof method, see chapter 3.3.2 for details).

When formal methods are used, some practices should be applied to facilitate the work of the evaluator and be more efficient.

- Formal models should be defined in accordance with some naming convention informations and is a huge help for traceability.

- Formal models should be define in accordance with "coding" rules ([JH08]). The proofs associated can be replay.

- Documentation and deliveries should respect templates and integrate traceability with requirements or elements from input specifications. From this point, the use of the Isabelle interface should be interesting with regard to its functionalities, refers to 2.3.5.

[7] to check than no parts of the code violate the security properties by side effects.

3.6 Summary

3.6.1 Background References

The most notable text describing the scientific history behind the LCF-family of **hol!** provers is done by by Mike Gordon[Gor00]. It covers the beginning of the entire research programme from 1972 to the mid-80ies, ranging from foundational issues of the logic over contributions to type-systems (as the "Hindley-Milner-Polymorphism")[Mil78] to the issue of the practical, safe implementation of rewrites and decision procedures [Pau99].

The LCF research programme was in parallel to another notable source of nowadays interactive theorem proving technologies: the Automath-project. In 1968, N.G. de Bruijn designs the first computer program to check the validity of general mathematical proofs, using typed λ-calculi as a direct means to represent proof objects as such. The emphasis of this programme was initially on proof-checking; de Bruijn's system Automath eventually checked every proposition in a primer that Landau had written for his daughter on the construction of real numbers as Dedekind cuts. A descendant of this family, which also has deeply influenced the Isabelle kernel design (proof objects, core inferences) is the Coq system (see http://coq.inria.fr).

Another notable survey on research programme is contained in the papers contained in *A Special Issue on Formal Proof* distributed by the American Mathematical Society (see http://www.ams.org/notices/200811/, but also [Hal08]), which presents nicely the relevance of modern ITP technology for purely mathematical problems (an argument, which has been strengthened recently by the formal proof of the Feit-Tompson theorem, whose precise formulation has haunted mathematicians for decades [Gon13], and the formal proof of the Kepler-conjecture, which is a known mathematical problem for about 400 years.).

3.6.2 Concluding Remarks and a Summary

We have presented the Isabelle/**hol!** system and pointed out the essential arguments, why by a particular combination of system-architecture and methodology, the system is suited to give the currently highest possible guarantee on a formal proof in particular and a logical theory development in general. In a sense, Isabelle/**hol!** offers the same guarantees for logical systems as Coq[JH08], and in some sense better guarantees than, for example, the B method or model-checkers like FDR. Isabelle/**hol!** is therefore a natural choice for evaluations in the higher certification levels EAL5 to EAL7 in the Common Criteria (CC) [Mem06].

If the methodological side-conditions are respected which can be reduced essentially to an number syntactic checks, the formal consistency of the entire

certification document containing formal specifications, proofs of consistency and the proofs of security properties, refinement-proofs between the different abstraction layers, and finally test-case generations as well as test-results can be guaranteed, and the evaluator can therefore concentrate on the more fundamental questions: does the model represent the right thing? are the modeling assumptions justified?

As "take-home-message" we would summarize these side-conditions as follows:

- Use a trusted, unmodified Isabelle version from the distribution.
- Check the restriction to definitional axioms only, enforce the use of "safe" specification constructs discussed here.
- Rule out `axiomatization`, `sorry`, their variants or disguised equivalents (such as oracle declarations).
- In particular `sorry`'s or equivalent constructions in methodological proofs have to be ruled out.
- Check the quick-and-dirty mode status.
- Exploring a TOE interactively, for example by jEdit, which allows for inspecting theories and definitions, their animation, the checking of types and of proof details, is a great means to increase confidence for an evaluator. However, the final check should be done in a non-interactive mode (pretty-printing and display machinery is actually quite far from the kernel and can be erroneous in itself).
- The main theorem in an CC evaluation is presumably of the form: "the security property X stated in the context of the security model Y is satisfied for the functional model Z under some conditions A in some locale B". A skeptical evaluator may insist on proofs that A and B are actually satisfiable, under circumstances even in a constructive sense.
- A conservative evaluator should restrict or ban `ML`-statements (with the possible exception of declarations of antiquotations), otherwise inspect `ML`-statements with particular care.

The internal code generator (also used in `code`-antiquotations and `value`-statements) stood the test of the time, but enjoys not quite the same level of trust as the proof facilities. The generation of proof objects for a complete theory is in principle possible, but should not be necessary except in case of a concrete suspicion of a fraudulent proof attempt.

4

Theoretical and Technical Foundations: Testing Concurrent Programs

Contents

4.1 Introduction

The verification of systems combining soft- and hardware, such as modern avionics systems, asks for combined efforts in test and proof: In the context of certifications such as EAL5 in Common Criteria section 3.2, the required formal security models have to be linked to system models via refinement proofs, and system models to code-level implementations via testing techniques.

Our work complements the testing initiative by a proof-based testing technique linking the formal system model of the PikeOS inter-process communication against the real system. This is a technical challenge for at least the following reasons:

- the system model is a transaction machine over a very rich state,
- system calls were implemented by internal, uninterruptible "atomic actions" reflecting the L4-microkernel concept; atomic actions define the granularity of our concurrency model, and
- the security model is complex and, in case of aborted system calls, leads to non-standard notions of execution trace interleaving.

To meet these challenges, we need to revise conceptual and theoretical foundations.

- We use symbolic execution techniques to cope with the large state-space; their inherent drawback to be limited to relatively short execution traces is outweighed by their expressive power,
- we extend the "monadic test approach" proposed in [BW07, BW13] to a test-method for concurrent code. It combines an IO-automata view [LT89a] with extended finite state machines [Gil62] using abstract transitions, and
- we need an adaption of concurrency notions, a "semantic view" on partial-order reduction and its integration into interleaving-based coverage criteria.

This sums up to a novel, tool-supported, integrated test methodology for concurrent OS-system code, ranging from an abstract system model in Isabelle/HOL, complemented embedding of the latter into our monadic sequence testing framework, our setups for symbolic execution down to generation of test-drivers and the code instrumentation.

In this chapter we will introduce a set of technical and theoretical contributions to test concurrent programs. On theoretical side, we present the monadic test approach from an IO-Automata view in section 4.2 then we

show how it can be used to express concurrent test scenarios in section 4.4. In section 4.3 we state our refinement relation, which help us to express a family of conformance relations to link the abstract model with the concrete implementation. On the technical side, we will show how Isabelle is used as an abstract test case generator in section 4.5. Finally, our techniques to build test drivers for concurrent code are presented in section 4.7.

4.2 Monads Theory

The obvious way to model the state transition relation of an automaton A is by a relation of the type $(\sigma \times (\iota \times o) \times \sigma)\,\text{set}$; isomorphically, one can also model it via:

$$\iota \Rightarrow (\sigma \Rightarrow (o \times \sigma)\,\text{set})$$

or for a case of a deterministic transition function:

$$\iota \Rightarrow (\sigma \Rightarrow (o \times \sigma)\,\text{option})$$

In a theoretic framework based on classical higher-order logic (HOL), the distinction between "deterministic" and "non-deterministic" is actually much more subtle than one might think: since the transition function can be under-specified via the Hilbert-choice operator, a transition function can be represented by

$$\mathit{step}\ \iota\ \sigma = \{(o, \sigma')|\ \text{post}(\sigma, o, \sigma')\}$$

or:

$$\mathit{step}\ \iota\ \sigma = \mathit{Some}(\text{SOME}(o, \sigma').\ \text{post}(\sigma, o, \sigma'))$$

for some post-condition post. While in the former "truly non-deterministic" case *step* can and will at run-time choose different results, the latter "under-specified deterministic" version will decide in a given model (so to speak: the implementation) always the same way: a choice that is, however, unknown at specification level and only declaratively described via post. For the system in this paper and our prior work on a processor model [BFNW13], it was possible to opt for an under-specified deterministic stepping function.

We abbreviate functions of type $\sigma \Rightarrow (o \times \sigma)\,\text{set}$ or $\sigma \Rightarrow (o \times \sigma)\,\text{option}$ $\text{MON}_{\text{SBE}}(o, \sigma)$ or $\text{MON}_{\text{SE}}(o, \sigma)$, respectively; thus, the aforementioned state transition functions of io-automata can be typed by $\iota \to \text{MON}_{\text{SBE}}(o, \sigma)$ for the general and $\iota \to \text{MON}_{\text{SE}}(o, \sigma)$ for the deterministic setting. If these function spaces were extended by the two operations *bind* and *unit* satisfying three algebraic properties, they form the algebraic structure of a *monad* that is well known to functional programmers as well as category theorists. Popularized by [Wad92], monads became a kind of standard means to incorporate stateful computations into a purely functional world.

Since we have an underspecified deterministic stepping function in our system model, we will concentrate on the latter monad which is called the *state-exception monad* in the literature.

The operations *bind*, which represent sequential composition with value passing, and *unit*, which represent the embedding of a value into a computation, are defined for the special-case of the state-exception monad as follows:

```
definition bind_SE :: ('o,'σ)MON_SE ⇒('o ⇒('o','σ)MON_SE) ⇒
                        ('o','σ)MON_SE
where      bind_SE f g = (λσ. case f σof None ⇒None
                                | Some (out, σ') ⇒g out σ')
```

```
definition unit_SE :: 'o ⇒('o, 'σ)MON_SE ((return _) 8)
where      unit_SE e = (λσ. Some(e,σ))
```

We will write $x \leftarrow m_1; m_2$ for the sequential composition of two (monad) computations m_1 and m_2 expressed by $\text{bind}_{\text{SE}}\, m_1 (\lambda\, x.m_2)$. Moreover, we will write "return" for unit_{SE}.

This definition of bind_{SE} and unit_{SE} satisfy the required monad laws:

```
lemma bind_left_unit [simp]:
  (x ←return c; P x) = P c
  by (simp add: unit_SE_def bind_SE_def)
```

```
  lemma bind_right_unit[simp]:
    (x ←m; return x) = m
    apply (simp add: unit_SE_def bind_SE_def)
    apply (rule ext)
    apply (case_tac m σ, simp_all)
    done
```

```
lemma bind_assoc[simp]:
  (y ←(x ←m; k x); h y) =
   (x ←m; (y ←k x; h y))
  apply (simp add: unit_SE_def bind_SE_def, rule ext)
  apply (case_tac m σ, simp_all)
  apply (case_tac a, simp_all)
  done
```

On this basis, the concept of a *valid monad execution*, written $\sigma \models m$, can be expressed: an execution of a Boolean (monad) computation m of type (bool, σ) MON_{SE} is valid if and only if its execution is performed from the initial state σ, no exception occurs and the result of the computation is true.

```
definition valid_SE ::
 'σ ⇒ (bool,'σ) MON_SE ⇒bool (infix ⊨15)
   where (σ ⊨m) = (m σ ≠ None ∧fst(the (m σ)))
```

More formally, $\sigma \models m$ holds if and only if $(m\ \sigma \neq \text{None} \wedge \text{fst}(\text{the}(m\ \sigma)))$, where `fst` and `snd` are the usual *first* and *second* projection into a Cartesian product and `the` is the projection in the `Some` a variant of the option type.

We define a *valid test-sequence* as a valid monad execution of a particular format: it consists of a series of monad computations $m_1 \ldots m_n$ applied to inputs $\iota_1 \ldots \iota_n$ and a post-condition P in a `return` depending on observed output. It is formally defined as follows:

$$\sigma \models o_1 \leftarrow m_1\ \iota_1; \ldots; o_n \leftarrow m_n\ \iota_n; \text{return}(P\ o_1 \cdots o_n)$$

The notion of a valid test-sequence has two facets: On the one hand, it is executable, i. e., a *program*, iff $m_1, \ldots, m_n, P$ are. Thus, a code-generator can map a valid test-sequence statement to code, where the m_i where mapped to operations of the SUT interface. On the other hand, valid test-sequences can be treated by a particular simple family of symbolic executions calculi, characterized by the schema (for all monadic operations m of a system, which can be seen as its step-functions):

$$\frac{}{(\sigma \models \text{return}\, P) = P} \tag{4.1a}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = None}{(\sigma \models ((s \leftarrow m\ \iota; m'\ s))) = False} \tag{4.1b}$$

$$\frac{C_m\ \iota\ \sigma \qquad m\ \iota\ \sigma = Some(b, \sigma')}{(\sigma \models s \leftarrow m\ \iota; m'\ s) = (\sigma' \models m'\ b)} \tag{4.1c}$$

Which corresponds to the following Isabelle/HOL implementation:

```
lemma exec_unit_SE [simp]: (σ ⊨(return P)) = (P)
  by(auto simp: valid_SE_def unit_SE_def)
```

```
lemma exec_bind_SE_failure:
  A σ = None ⟹¬(σ ⊨((s ←A ; M s)))
  by(simp add: valid_SE_def unit_SE_def bind_SE_def)
```

```
lemma exec_bind_SE_success:
  A σ = Some(b,σ') ⟹(σ ⊨((s ←A ; M s))) = (σ' ⊨(M b))
  by(simp add: valid_SE_def unit_SE_def bind_SE_def )
```

This kind of rules is usually specialized for concrete operations m; if they contain pre-conditions C_m (constraints on ι and state), this calculus will just accumulate those and construct a constraint system to be treated by constraint solvers used to generate concrete input data in a test.

4.2.1 An Example: MyKeOS.

To present the effect of the symbolic rules during symbolic execution, we present a toy OS-model. MyKeOS provides only three atomic actions for *allocation* and *release* of a resource (for example a descriptor of a communication channel or a file-descriptor). A *status* operation returns the number of allocated resources. All operations are assigned to a thread (designated by `thread_id`) belonging to a task (designated by `task_id`, a Unix/POSIX-like *process*); each thread has a thread-local counter in which it stores the number (the status) of the allocated resources. The input is modeled by the data-type:

```
datatype in_c = alloc task_id thread_id nat
              | release task_id thread_id nat
              | status task_id thread_id
```

```
datatype out_c = alloc_ok | release_ok | status_ok nat
```

where `out_c` captures the return-values. Since `alloc` and `release` do not have a return value, they signalize just the successful termination of their corresponding system steps. The global table `var_tab` (corresponding to our symbolic state σ) of thread-local variables is modeled as partial map assigning to each active thread (characterized by the pair of task and thread id) the current status:

```
type_synonym thread_local_var_tab = (task_id ×thread_id) ⇀int
```

The operation have the precondition that the pair of task and thread id is actually defined and, moreover, that resources can only be released that have been allocated; the initial status of each defined thread is set to 0. The **hol!** representation of the preconditions and post-conditions is:

```
fun precond :: thread_local_var_tab ⇒in_c ⇒bool
where
 precond σ (alloc taskid thid res) = ((taskid,thid) ∈dom σ)
| precond σ (release taskid thid res) = ((taskid,thid) ∈dom σ∧
                                  (int res) ≤the(σ(taskid,thid)))
| precond σ (status taskid thid) = ((taskid,thid) ∈dom σ)
```

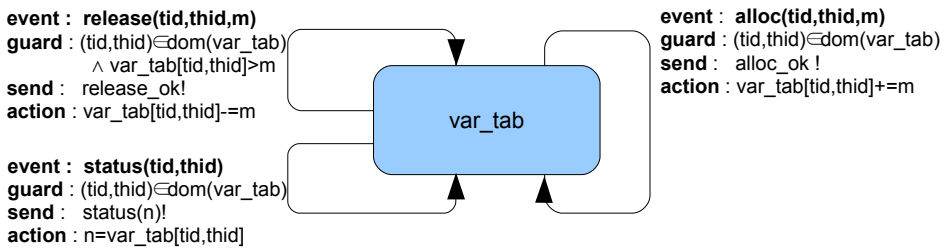


Figure 4.1: SPEC: An Extended Finite State Machine for MyKeOS.

```
fun postcond :: in_c ⇒thread_local_var_tab ⇒
                (out_c ×thread_local_var_tab) set
where
 postcond (alloc taskid thid res) σ=
     { (n,σ'). (n = alloc_ok ∧
                σ'=σ((taskid,thid)↦ the(σ(taskid,thid)) + int res))}
| postcond (release taskid thid res) σ=
   {(n,σ'). (n = release_ok ∧
             σ'=σ((taskid,thid)↦ the(σ(taskid,thid)) - int res))}
| postcond (status taskid thid) σ=
   {(n,σ'). (σ=σ' ∧
            (∃ x. status_ok x = n ∧x = nat(the(σ(taskid,thid)))))}
```

Depicted as an extended finite state-machine (EFSM), the operations of our system model SPEC are specified as shown in Figure 4.1.

A transcription of an EFSM to HOL is scattered here[1]. Actually the HOL model of an EFSM is represented by a locale[Bal10], which is instantiated by the above definitions of pre-post conditions and:

```
 definition strong_impl ::
 ['σ⇒'ι⇒bool, 'ι⇒('o,'σ)MON_SB] ⇒'ι⇒('o, 'σ)MON_SE
 where strong_impl pre post ι=
       (λ σ. if pre σ ι
             then Some(SOME(out,σ'). (out,σ') ∈post ισ)
             else None)
```

```
definition SPEC = (strong_impl precond postcond)
```

where SPEC represent the instantiation of an EFSM with the semantics of MyKeOS. We show a concrete symbolic execution rule derived from the definitions of the SPEC system transition function, e. g., the instance for Equa-

[1]It is a theory in HOL-TestGen distribution.

tion 4.1c:

$$\frac{(tid, thid) \in \mathrm{dom}(\sigma) \qquad \text{SPEC (alloc } tid\ thid\ m)\ \sigma = Some(\text{alloc\_ok}, \sigma')}{(\sigma \models s \leftarrow \text{SPEC (alloc } tid\ thid\ m); m'\ s) = (\sigma' \models m'\ \text{alloc\_ok})}$$

where $\sigma = var\_tab$ and $\sigma' = \sigma((tid, thid) := (\sigma(tid, thid) + m))$. Thus, this rule allows for computing σ, σ' in terms of the free variables var_tab, tid, $thid$ and m. The rules for release and status are similar. For this rule, SPEC (alloc $tid\ thid\ m$) is the concrete stepping function for the input event alloc $tid\ thid\ m$, and the corresponding constraint C_{SPEC} of this transition is $(tid, thid) \in \mathrm{dom}(\sigma)$.

4.3 Conformance Relations Revisited

We state a family of test conformance relations that link the specification and abstract test drivers. The trick is done by a coupling variable *res* that transport the result of the symbolic execution of the specification SPEC to the expected result of the SUT.

$$\begin{aligned}&\sigma \models o_1 \leftarrow \text{SPEC}\ \iota_1; \ldots; o_n \leftarrow \text{SPEC}\ \iota_n; \text{return}(res = [o_1 \cdots o_n])\\ &\longrightarrow\\ &\sigma \models o_1 \leftarrow \text{SUT}\ \iota_1; \ldots; o_n \leftarrow \text{SUT}\ \iota_n; \text{return}(res = [o_1 \cdots o_n])\end{aligned}$$

Successive applications of symbolic execution rules allow to reduce the premise of this implication to $C_{\text{SPEC}}\ \iota_1\ \sigma_1 \longrightarrow \ldots \longrightarrow C_{\text{SPEC}}\ \iota_n\ \sigma_n \longrightarrow res = [a_1 \cdots a_n]$ (where the a_i are concrete terms instantiating the bound output variables o_i), i. e., the constrained equation $res = [a_1 \cdots a_n]$. The latter is substituted into the conclusion of the implication. In our previous example, case-splitting over input-variables ι_1, ι_2 and ι_3 yields (among other instances) $\iota_1 = \text{alloc}\ t_1\ th_1\ m$, $\iota_2 = \text{release}\ t_2\ th_2\ n$ and $\iota_3 = \text{status}\ t_3\ th_3$, which allows us to derive automatically the constraint:

$$\begin{gathered}(t_1, th_1) \in \mathrm{dom}(\sigma) \longrightarrow\\ (t_2, th_2) \in \mathrm{dom}(\sigma') \wedge n < \sigma'(t_2, th_2) \longrightarrow\\ (t_3, th_3) \in \mathrm{dom}(\sigma'') \longrightarrow res = [\text{alloc\_ok}, \text{release\_ok}, \text{status\_ok}(\sigma''(t_3, th_3))]\end{gathered}$$

where $\sigma' = \sigma((t_1, th_1) := (\sigma(t_1, th_1) + m))$ and $\sigma'' = \sigma'((t_2, th_2) := (\sigma(t_2, th_2) - n))$. In general, the constraint $C_{\text{SPEC}_i}\ \iota_i\ \sigma_i$ can be seen as an *symbolic abstract test execution*; instances of it (produced by a constraint solver such as Z3 integrated into Isabelle) will provide concrete input data for the valid test-sequence statement over SUT, which can therefore be compiled to test driver code. In our example here, the witness $t_1 = t_2 = t_3 = 0$, $th_1 = th_2 = th_3 = 5$, $m = 4$ and $n = 2$ satisfies the constraint and would produce (predict) the

output sequence $res = [\text{alloc\_ok}, \text{release\_ok}, \text{status\_ok}\,2]$ for SUT according to SUT. Thus, a resulting (abstract) test-driver is:

$$\begin{aligned}&\sigma \models o_1 \leftarrow \text{SUT}\ \ \iota_1; \ldots; o_3 \leftarrow \text{SUT}\ \ \iota_3;\\ &\text{return}([\text{alloc\_ok}, \text{release\_ok}, \text{status\_ok}\,2] = [o_1 \cdots o_3])\end{aligned}$$

This schema of a test-driver synthesis can be refined and optimized. First, for iterations of stepping functions an 'mbind' operator can be defined, which is basically a fold over bind_{SE}. It takes a list of inputs $\iota s = [i_1, \ldots, i_n]$, feeds it subsequently into SPEC and stops when an error occurs. Using mbind, valid test sequences for a stepping-function (be it from the specification SPEC or the SUT) evaluating an input sequence ιs and satisfying a post-condition P can be reformulated to:

$$\sigma \models os \leftarrow \text{mbind}\ \ \iota s\, \text{SPEC}; \text{return}(P\ os)$$

Second, we can now formally define the concept of a test-conformance notion:

$$\begin{aligned}&(\text{SPEC} \sqsubseteq_{\langle \text{Init,CovCrit,conf} \rangle} \text{SUT}) =\\ &(\forall \sigma_0 \in \mathit{Init}.\ \forall \iota\ s \in \mathit{CovCrit}.\ \forall \mathit{res}.\\ &\sigma_0 \models os \leftarrow \text{mbind}\ \ \iota s\ \ \text{SPEC};\ \ \text{return}(\mathit{conf}\ \ \iota s\ \ os\ \ \mathit{res})\\ &\longrightarrow \sigma_0 \models (os \leftarrow \text{mbind}\, \iota s\ \ \mathit{SUT};\ \ \text{return}(\mathit{conf}\ \ \iota s\ \ os\ \ \mathit{res})))\end{aligned}$$

For example, if we instantiate the conformance predicate conf by:

$$\text{conf}\ \ \iota s\ \ os\ \ \mathit{res} = (\text{length}(\iota s) = \text{length}(os) \wedge \mathit{res} = os)$$

we have a precise characterization of inclusion conformance introduced in subsection 2.2.3: We constrain the tests to those test sequences where no exception occurs in the symbolic execution of the model. Symbolic execution fixes possible output-sequence (which must be as long as the input sequence since no exception occurs) in possible symbolic runs with possible inputs, which must be exactly observed in the run of the SUT in the resulting abstract test-driver.

Using pre and post-condition predicates, it is straight-forward to characterize deadlock conformance or IOCO mentioned earlier (recall that our framework assumes synchronous communication between tester and SUT; so this holds only for a IOCO-version without quiescence). Further, we can characterize a set of initial states or express constraints on the set of input-sequences by the *coverage criteria CovCrit*, which we will discuss in the sequel.

4.4 Coverage Criteria for Interleaving

In the following, we consider input sequences ιs which were built as interleaving of one or more inputs for different processes; for the sake of simplicity,

we will assume that it is always possible to extract from an input event the thread and task id it belongs to. It is possible to represent this interleaving, for example, by the following definition:

```
fun interleave :: 'a list ⇒'a list ⇒'a list set
where interleave [] [] = {[]}
     |interleave A [] = {A}
     |interleave [] B = {B}
     |interleave (a # A) (b # B) =
        image (λx. a # x) (interleave A (b # B)) ∪
        image (λx. b # x) (interleave (a # A) B)
```

and by requiring for the input sequence ιs to belong to the set of interleaving of two processes `P1` and `P2`: ι`s` ∈`interleave P1 P2`. It is well known that the combinatorial explosion of the interleaving space represents fundamental problem of concurrent program verification. Testing, understood as the art of creating finite, well-chosen sub-spaces for large input-output spaces, offers solutions based on adapted coverage criteria [SLZ07] of these spaces, which refers to particular instances of `CovCrit` in the previous section. A well-defined coverage criterion [ZHM97, FTW04] can reduce a large set of interleaving to a smaller and manageable one. For example, consider the executions of the two threads in MyKeOS: `T = [alloc 3 1 2, release 3 1 1, status 3 1]` and `T' = [alloc 2 5 3, release 3 1 1, status 2 5]`. Since our simplistic MyKeOS has no shared memory, we simulate the effect by allowing `T'` to execute a `release`-action on the local memory of task 3, thread 1 by using its identity. In general, we are interested in all possible values of a shared program variable x at position l after the execution of a process P. To this end we will define two sets of interleaving under two different known criteria.

- **Criterion1: standard interleaving** (SIN) *the interleaving space of actions sequences gets a complete coverage iff all feasible interleaving of the actions of P are covered.*
- **Criterion2: state variable interleaving** (SVI) *the interleaving space of actions sequences gets a complete coverage iff all possible states of x at l in P are covered.*

Under SIN we derive 10 possible actions sequences, which is reduced under SVI to 3 sequences (where one leads to a crash; recall our assumption that the memory is initially 0). Unlike to SIN, SVI has provided a smaller interleaving set that cover all possible states. If we consider `var_tab[3,1]` for x when executing `status 3 1`, the possible results may be undefined, O or 1. While SIN has provided a bigger set, that cover all possible 3 states of x with redundant sequences representing the same value. In model-checking, this reduction technique is also known as partial order reduction [Pel93, GW94].

It is a part of a beauty for our test and proof approach, that we can actually formally prove that the test-sets resulting from the test-refinements:

$$\text{SPEC} \sqsubseteq_{\langle Init,SIN,conf\rangle} \text{SUT} \quad \text{and} \quad \text{SPEC} \sqsubseteq_{\langle Init,SVI,conf\rangle} \text{SUT}$$

are equivalent for a given SPEC. The core of such an equivalence proof is, of course, a proof of commutativity of certain step executions, so properties of the form:

$$o \leftarrow \text{SPEC}\ \iota_i; o' \leftarrow \text{SPEC}\ \iota_j; M\ o\ o' = o' \leftarrow \text{SPEC}\ \iota_j; o \leftarrow \text{SPEC}\ \iota_i; M\ o\ o',$$

which are typically resulting from the fact that these executions depend on disjoint parts of the state. In MyKeOS, for example, such a property can be proven automatically for all $\iota_i = \text{release}\ t\ th$ and $\iota_j = \text{release}\ t'\ th'$ with $t \neq t' \vee th \neq th'$; such reordering theorems justify a partial order on inputs to reduce the test-space. We are implicitly applying the testability hypothesis that SUT is input-output deterministic; if a input-output sequence is possible in SPEC, the assumed input-output determinism gives us that repeating the test by an equivalent one will produce the same result.

4.5 Sequence Test Scenarios for Concurrent Programs

HOL-TestGen is a test-generation system based on the Isabelle theorem prover. The main goal of this system is to use the features of Isabelle in order to generate a test set. Using the isar command `test_spec` from HOL-TestGen framework a test scenario can be represented in form of a test specification. A test specification is an **hol!** formula, i. e. a valid test sequence, that describe the test set to be generated. Two possible schemes for a test scenario can be expressed by a test specification: *unit test scheme, sequence test scheme.* In this section we will focus on sequence test scenarios. In sequence test scenarios, a set of input sequences are generated under a given coverage criteria and symbolically executed (see section 4.6 for more details on our symbolic execution process). Actually, a test specification is a `lemma` which contains our refinement relation (see section 4.3) as a proof statement. The representation of a refinement relation, for a scenario related to MyKeOS system, using Isabelle/isar language can be:

```
1  test_spec test_status:
2  assumes account_defined: (tid,0) ∈dom σ0∧(tid,1) ∈dom σ0
3  and     CovCrit  : S ∈interleave (syscall tid 0 m m')
4                                   (syscall tid 1 m'' m''')
5  and     SPEC :
6          σ 0 ⊨(s ←mbind S SPEC; return (x = s))
7  shows   σ 0 ⊨(s ←mbind S PUT; return (x = s))
```

In the scenario `test_status` the assumption `account_defined` is used to bound the set of threads to 2 members in each task and at least a task exists in the system. The assumption `CovCrit` represent the set of possible input sequences related to the concurrent execution between `syscall tid 0 m m'` and `syscall tid 1 m'' m'''`. Moreover, `SPEC` represent the model of the behaviour of the SUT. Finally, the conclusion $\sigma_0 \models$`(s ←mbind S PUT ; return (s = x))` is used to link the model with the real system via the free variable `PUT`. Actually, the free variable `PUT` will be linked to the actual code of SUT during the execution of the *test script* (for more details linkage between a model and a SUT see section 4.7).

In fact, the representation of `test_status` by a `lemma` offers a way to use the symbolic computation engine of Isabelle, usually used for proofs, as a simulation environment for the behaviour of the SUT. Basically, the simulation is done via the application of symbolic execution rules, e. g. an instance for Equation 4.1c, on the proof statement, which result with a set of sub-goals. Each sub-goal represent an abstract test case, and each abstract test case is a representation of a set of possible executions in the SUT.

The simulation, related to the behavior of MyKeOS specified in the scenario `test_status`, using symbolic execution on Isabelle, is represented by the following:

```
1
2     (...)
3     (*************************************
4      ***Resulting proof statement: ctxt1***
5      *************************************)
6     1. σ 0 ⊨( s ←mbind [alloc tid 1 m'', release tid 0 m',
7                         release tid 1 m''', status tid 1]
8                   SYS; unit_SE (x = s)) ⟹
9       σ 0 ⊨( s ←mbind S PUT; unit_SE (s = x))
```

```
1       (*****************************
2        ***rules applied on: ctxt1***
3        *****************************)
4       apply(tactic ematch_tac [@{thm status.exec_mbindFStop_E},
5                                @{thm release.exec_mbindFStop_E},
6                                @{thm alloc.exec_mbindFStop_E},
7                                @{thm H1}] 1)
```

```
1     (************************************
2      ***Resulting proof statement: ctxt2***
3      ************************************)
4     1.(tid, 1) ∈dom σ 0 ⟹
5       σ 0((tid, 1) ↦the (σ 0 (tid, 1)) + int m'') ⊨
6       (s ←mbind [release tid 0 m', release tid 1 m''', status tid 1]
7                 SYS ; unit_SE (x = alloc_ok # s)) ⟹
8     σ 0 ⊨( s ←mbind S PUT; unit_SE (s = x))
```

```
1     (*****************************
2      ***rules applied on: ctxt2***
3      *****************************)
4     apply(tactic ematch_tac [@{thm status.exec_mbindFStop_E},
5                              @{thm release.exec_mbindFStop_E},
6                              @{thm alloc.exec_mbindFStop_E},
7                              @{thm H1}] 1)
```

```
1     (************************************
2      ***Resulting proof statement: ctxt3***
3      ************************************)
4     1. (tid, 1) ∈dom σ 0 ⟹
5        (tid, 0) ∈dom (σ 0((tid, 1) ↦the (σ 0 (tid, 1)) + int m'')) ∧
6        int m' ≤
7        the ((σ 0((tid, 1) ↦the (σ 0 (tid, 1)) + int m'')) (tid, 0)) ⟹
8        σ 0((tid, 1) ↦the (σ 0 (tid, 1)) + int m'', (tid, 0) ↦
9        the ((σ 0((tid, 1) ↦the (σ 0 (tid, 1)) + int m'')) (tid, 0)) -
10       int m') ⊨
11       ( s ←mbind [release tid 1 m''', status tid 1]
12                  SYS ; unit_SE (x = alloc_ok # release_ok # s)) ⟹
13     σ 0 ⊨( s ←mbind S PUT; unit_SE (s = x))
14    (...)
```

A such *proof context refinement process*, is executed until the input sequence of actions is empty, which provide directly for the case of a test specification of a simple operational semantics, a *test normal forms*, represented by subgoals. Of course, the proof statement can be connected to constraint-solvers with the HOL-TestGen command `gen_test_data`, which will instantiate the free variables,e. g. σ_0, `tid` in the different subgoals of the proof statement, by a real data that satisfies the derived constraints.

4.6 Symbolic Execution

Symbolic execution rules, are logical inference rules used to simulate the behavior of a given system (or a program) by showing the effect of the operational semantics of that system (or program) on the *symbolic variables*. Symbolic variables are a typed syntactic names used to refer to a given object (i. e. a passive entity in the operating system), that may have an infinite set of representations (values). In general, two kind of variables are distinguished in an operating system, *global variables* and *local variables*. For instance, in our test specification `test_status` the variable σ_0 can be seen as a global variable that refer to the state of the system (i. e. an object which can be modified by all subjects (threads)).

In order to give a better explanation on the symbolic execution rules used during the simulation of the behavior of MyKeOS we would introduce the generic scheme of their **hol!** representation:

```
1  lemma exec_mbindFStop_E:
2  assumes A:(σ ╞=(s ←mbind (in_ev # S) efsm; return (P s)))
3  and     B:E σ ⟹
4             ((upd σ) ╞=(s←mbind S efsm;return(P(out_ev σ# s)))) ⟹
5             Q
6  shows    Q
7    by(insert A, rule B, simp_all del: mbind'_bind)
```

Code 1: A Generic Elimination Rule For Symbolic Execution

If we observe more closely the previous inference rule, we can figure out that the rule is an elimination rule. An elimination rule is an inference rule that eliminate a given constructor from the premises, i. e. in the rule `exec_mbindFStop_E` we had eliminated `in_ev` from the input sequence `(in_ev # S)`. Actually, the scheme of an elimination rule matches with the scheme of our test specifications, i. e. the free variable `Q` in `exec_mbindFStop_E` will match with σ_0 `╞=(s ←mbind S PUT; return (s = x))` in `test_status`; the assumption `A` of `exec_mbindFStop_E` will match with the assumption `SPEC` of `test_status`, and the resulting proof context after the application of this

elimination inference rule on the test specification `test_status` will be, the instatiation of the assumption `B` of `exec_mbindFStop_E` by the variables of `SPEC`. A such process is used for transforming the proof context, and it is calles *ematching*. On Isabelle ematching can be expressed by the tactic `ematch_tac`.

Our symbolic execution process, which is actually based on proof context transformations by ematching, has an enormous performance gain effect on the symbolic execution engine of Isabelle (see subsection 6.5.7 for the impressive results of a such process of symbolic execution). The performance gain provided by our process is comming from the fact that, the whole calculation process is technically reduced to a *formal* syntactic transformation of the proof context by elimination rules (applied by ematching), instead of a calculation process based on standard generic substitution rules, rewriting rules, introduction rules, etc., which involve more calculations in the different Isabelle layers.

4.7 Test Drivers for Concurrent C Programs

The generation of the test-driver is a non-trivial exercise since it is essentially two-staged: Firstly, we choose (from the different options the Isabelle code-generator offers) to generate an SML test-driver, which is then secondly, compiled to a C program that is linked to the actual program under test. A test-driver for HOL-TestGen consists of four components:

- `main.sml` the global controller (a fixed element in the library),
- `harness.sml` a statistic evaluation library (a fixed element in the library),
- `X_script.sml` the test-script that corresponds merely one-to-one to the generated test-data (generated)
- `X_adapter.sml` a hand-written program; in our scenario, it replaces the usual (black-box) program under test by SML code, that calls the external C-functions via a foreign function interface.

On all three levels, the HOL-level, the SML-level, and the C-level, there are different representations of basic data-types possible; the translation process of data to and from the C-code under test has therefore to be carefully designed (and the sheer space of options is sometimes a pain in the neck). Integers, for example, are represented in two ways inside Isabelle/HOL; there is the mathematical quotient construction and a "numerals" representation providing "bit-string-representation-behind-the-scene" enabling relatively efficient symbolic computation on integers. Both representations can be compiled "natively" to data types in the SML level. By an appropriate configuration, the code-generator can map "int" of HOL to three different implementations: the SML standard library `Int.int`, the native-

C interfaced by `Int32.int`, and the `IntInf.int` from the multi-precision library `gmp` underneath the polyml-compiler. We do a three-step compilation of data-representations Model-to-Model, Model-to-SML, SML-to-C. A basic preparatory step for the initializing the test-environment to enable test-generation is:

```
test_spec test_status2:
assumes system_def : (c0,no) ∈dom σ0
and     store_finite : σ0 = map_of T
and     test_purpose : test_purpose [(c0,no),(c0,no')] S
and     sym_exec_spec :
        σ0 ⊨(s ←mbind' S SYS; return (s = x))
shows   σ0 ⊨(s ←mbind' S PUT; return (s = x))
apply(rule rev_mp[OF sym_exec_spec])
apply(rule rev_mp[OF system_def])
apply(rule rev_mp[OF test_purpose])
apply(rule_tac x=x in spec[OF allI])
apply(gen_test_cases 3 1 PUT)
apply(auto intro: P1'' P2'')
store_test_thm mykeos_simple
gen_test_data mykeos_simple
generate_test_script mykeos_simple
```

The tool `store_test_thm` is a tool from HOL-TestGen framework. This tool provide the ability to users to store a given proof context of the test specification and refer to this proof context by a label (i. e. `mykeos_simple`). The tool `gen_test_data` from HOL-TestGen provide the ability to users to instantiate the symbolic variables inside abstract test cases by concrete data. The latter step is done by sending *proof obligations*, i. e. constraints on the variables generated during the symbolic execution, to constraint solvers in order to instantiate them with satisfiable witnesses. The tool `generate_test_script` is provided by HOL-TestGen framework. Basically, the tool provide the ability to users to transform the proof context stored using `store_test_thm` to a *code equation*; code equations are rewriting rules used as inputs for Isabelle code generators. For instance, the following code equation is resulting from the application of `gen_test_script` on the proof context labeled by the name `mykeos_simple`:

```
   mykeos_simple.test_script ≡
[([], lazy ((λa. Some -1) ⊨
    ( s ←mbind [alloc 3 5 (nat 2), status 3 5]
          PUT; unit_SE (s = [alloc_ok, status_ok (nat 1)])))),
 ([], lazy ((λa. if a = (2, 3) then Some 8465 else Some 8) ⊨
     (s ←mbind [release 2 3 (nat 8466), status 2 3] PUT;
      unit_SE (s = [])))),
 ([], lazy ((λa. Some 8468) ⊨
     ( s ←mbind [release 2 3 (nat 1), status 2 3]
          PUT; unit_SE (s = [release_ok, status_ok (nat 8467)])))),
 ([], lazy ((λa. if a = (2, 3) then Some 8465 else Some 8) ⊨
    ( s ←mbind [release 2 3 (nat 8466), status 2 3] PUT;
      unit_SE (s = [])))),
 ([], lazy ((λa. Some -1) ⊨
    ( s ←mbind [alloc 2 3 (nat 1), alloc 2 3 (nat 1), status 2 3]
          PUT;
     unit_SE (s = [alloc_ok, alloc_ok, status_ok (nat 1)])))),
   (...)]
```

4.7.1 The adapter

In the following, we describe the interface of the SML-program under test, which is in our scenario an *adapter* to the C code under test. This is the heart of the Model-to-SML translation. Actually, during the execution of the test script, the free variable specified inside the test specification under name `PUT` will be replaced by an adapter. In fact, the adapter is a function defined on the HOL-level, and its semantic is based on constant definitions called *stubs*. The stubs are replaced later-on by the semantic of the implementation using code serialisation technique offered by the interface of Isabelle code generator to link the Model-level with SML-level (see subsection 4.7.2 for technical details). Then we use a foreign function interface provided by `MLton` compiler to link SML-level to C-level(see subsection 4.7.3 for technical details). The HOL-level stubs for testing MykeOS are declared as follows:

```
(*The definition of the stubs*)
consts    status_stub :: task_id ⇒int ⇒(int, 'σ)MON_SE
consts    alloc_stub :: task_id ⇒int ⇒int ⇒(unit, 'σ)MON_SE
consts    release_stub:: task_id ⇒int ⇒int ⇒(unit, 'σ)MON_SE
```

On the Model-to-Model level, we provide a global step function that distributes to individual interface functions via stubs (mapped via the code generation to SML ...). This translation also represents uniformly nat by int's.

```
fun  stepAdapter :: (in_c ⇒(out_c, 'σ)MON_SE)
where
    stepAdapter(status tid thid) =
     (x ←status_stub tid thid; return(status_ok (my_nat_conv x)))
  | stepAdapter(alloc tid thid amount) =
     (_ ←alloc_stub thid thid (int amount); return(alloc_ok))
  | stepAdapter(release tid thid amount)=
     (_ ←release_stub tid thid (int amount); return(release_ok))
```

The `stepAdapter` function links the HOL-world and establishes the logical link to HOL stubs which were mapped by the code-generator to adapter functions in SML, which call internally to C-code inside `X_adapter.sml` via a Foreign Function Interface (FFI).

4.7.2 Code generation and Serialisation

In order to generate concrete code from our theories we will use the code generator [Haf15] facilities of Isabelle/HOL. It allows to turn a certain class of HOL specifications into corresponding executable code in a target language (e. g. SML). In this section, we will show how we build a setup to generate SML file containing our test script. As an example we will continue to run MykeOS example via the test specification `test_status2`.

In the first place, we will generate 2 SML files. The first one containing all datatypes used in our test specification. The second one containing an adapter for the variable representing the system under test called `PUT` in the test specification `test_status2`. Therefore, both files will be used as libraries for the test script and help to increase its readability. Using Isabelle "serialiser", we configure the code-generator to identify the `PUT` with the generated SML code implicitly defined by the above `stepAdapter` definition.

```
 (*Code Setup for Datatypes*)

(* Setup for input actions *)
code_printing
  type_constructor in_c => (SML) Datatypes.in_c
    |constant alloc => (SML) !(Datatypes.Alloc ( _ , _ , _))
    |constant release => (SML) !(Datatypes.Release ( _ , _ , _ ))
    |constant status => (SML) !(Datatypes.Status ( _ , _ ))
```

```
(* Setup for the outputs *)
code_printing
  type_constructor out_c => (SML) Datatypes.out'_c
    |constant alloc_ok => (SML) Datatypes.Alloc'_ok
    |constant release_ok => (SML) Datatypes.Release'_ok
    |constant status_ok => (SML) !(Datatypes.Status'_ok ( _ ))
```

Basically, the link between the stubs in HOL world and the SML functions that calls to the C is done by asking Isabelle code generator to replace the stubs by functions inside a given SML file. Technically this step is resumed by:

```
(*Serialisation: replacing the HOL stubs by actual semantics
                represented on SML-level*)
code_printing
constant status_stub  (SML MyKeOSAdapter.status)

code_printing
constant alloc_stub   (SML MyKeOSAdapter.alloc)

code_printing
constant release_stub (SML MyKeOSAdapter.release)
```

By the same technique we ask the code generator to replace the constant `PUT` by the function `stepAdapter`. The latter function, can be generated automatically, as we will see in the last step, and it contains the calls to the stubs which are now SML functions:

```
(*Serialisation: Linking the free variable PUTwith
                the concrete SML-code via stepAdapter*)
code_printing
constant PUT=> (SML) stepAdapter
```

And there we go and generate the `mykeos_simple`:

```
export_code          stepAdapter mykeos_simple.test_script in SML
module_name TestScript file impl/c/mykeos_simple_test_script.sml
```

4.7.3 Building Test Executables

Inside the SML file containing the module `adapter.sml`, we will use again serialisation technique via the compiler `MLton`. Actually, `MLton` provides a foreign function interface to C, this interface is used to call the actual semantic

of the program under test. `MLton` compiler provide a command to build the test executable for our generated test script in SML language, containing called function from the implementation in C language.

4.7.4 GDB and Concurrent Code Testing

Actually the generated build from `MLton` compiler will contain tests for threads executed in a concurrent context. The problem with executing tests on concurrent code is that: the execution order of the program actions proposed by the system scheduler will not necessarily be the same as the one proposed by the tester, and this because of the non-deterministic choices of the system scheduler. In order to deal with this problem, we have to enforce a certain order for the actions executed by the threads, i. e. a certain scheduling, during test execution. In other words, at run-time, the execution order proposed by the system scheduler must correspond to the execution order provided by the tester. Our solution is, the execution of the test executable within a GDB[2] session. The latter contain features, usually used for debugging, that can be used to control the execution of the concurrent code and make it conform to the generated executions proposed by the tester.

Technically, the gdb features allow the possibility to attach to breack-points within the concurrent code a scripting code that is executed if a break point is reached, and the complete control of thread switches. In order to generate automatically the GDB script that controls an execution of a system under test during a test experience, we had implemented a GDB generator on top of Isabelle/ML. Basically, the generator takes as argument 4 mandatory entries:

1. A function that setup the *entry breakpoints switches*,
2. A function that setup the *exit breakpoints switches*,
3. A function that setup the *main breakpoint switch*,
4. and a list of *needed informations* containing: thread IDs of the model with a mapping to there creation order inside a gdb session, and informations on the lines numbers for the breakpoints within the concurrent code.

Executed together, these functions implement an algorithm that setup thread switches for the program under test, that is conform to the switches represented by the generated input sequences from a test scenario on the model-level. Moreover note, the algorithm works correctly only if the single-core execution options are activated during the gdb session. In order to execute correctly the generated GDB scripts, the option `taskset` should be stated. It specifies

[2] https://www.gnu.org/software/gdb/

a single core execution for the gdb session. An abstraction of our algorithm used to generate GDB scripts from a test script is introduced in algorithm 1.

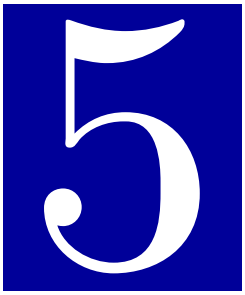
4.8 Conlusions

In this chapter we have presented our major contribution during this thesis. The chapter contains theoretical and technical foundations to test C concurrent program. On the theoretical side, we had presented our test generation framework which relies on a monadic test theory implemented in Isabelle/**hol!**. Our framework is equipped with a specification language based on monads that contains important definitions for testing and symbolic execution activities. First, in order to show the expressive power of our specification language, an isomorphism between the automata world and monads world was presented. Second, in order to provide a generic framework to express state exception behavior, two monad operators were introduced `bind_SE` and `unit_SE`. Based on the latter operators, a new concept called valid test sequence was defined. On the one hand, the notion of a valid test sequence is used to express the behavior of a given system. On the other hand, it is executable and can be treated by a family of symbolic executions calculi. A set of generic symbolic execution rules, for the defined operators, were introduced and in order to show how these concepts are used to model and/or to symbolically execute a given system, a running example on a simple OS called MykeOS was presented. Third, we proposed a generic scheme called test specification, expressed technically by a refinement relation, to link a specification with an implementation, then we had showed how it can be instantiated with a family of test conformance relations. Finally, in order to optimize the symbolic execution process for our test specifications, especially for the case of sequence test scenarios, an approach based on the notion of coverage criteria was proposed.

On the technical side, we highlighted the problem of testing concurent programs, and we proposed an approach to test concurrent C code using scheduler control via GDB scripts. Finally, we had showed how Isabelle/**hol!** can easily supports and carry our tools, going from symbolic execution on **hol!** down to test script on code level.

```
set entry breakpoints switches;
if input_seq ≠ empty and
find_thread_ID (info_thread) (input_seq) = i then
    switch to thread i
else
    if input_seq = empty then
        error empty input sequence
    else
        error thread informations are wrong
    end
end
set exit breakpoints switches;
if input_seq ≠ empty and
find_thread_ID (info_thread) (input_seq) = i and
next(input_seq) ≠ empty then
    switch to the successor of thread i
else
    if next(input_seq) = empty then
        switch to thread i
    else
        if input_seq = empty then
            error empty input sequence
        else
            error thread informations are wrong
        end
    end
end
set main breakpoint switch;
if input_seq ≠ empty and
find_thread_ID (info_thread) (head (input_seq)) = i then
    switch to thread i
else
    if input_seq = empty then
        error empty input sequence
    else
        error thread informations are wrong
    end
end
```

Algorithm 1: A pseudo code for the Switches Between Threads Forced by The GDB Script



Testing VAMP Processor

Contents

5.1 Introduction

Certifications of critical security or safety system properties are becoming increasingly important for a wide range of products. Certifying large systems like operating systems up to Common Criteria EAL 4 is common practice today, and higher certification levels are at the brink of becoming reality.

To reach EAL 7 one has to formally verify properties on the specification as well as test the implementation thoroughly. This includes tests of the used hardware platform of the architecture to be certified. In this chapter, we address the latter problem: we present a case study that uses a formal model of a microprocessor and generate test programs from it. These test programs validate that a microprocessor implements the specified instruction set correctly.

We built our case study on an existing model that was, together with an operating system, developed in Isabelle/HOL. We use HOL-TestGen, a model-based testing environment which is an extension of Isabelle/HOL. We develop several conformance test scenarios, where processor models were used to synthesize test programs that were run against real hardware in the loop. Our test case generation approach directly benefits from the existing models and formal proofs in Isabelle/HOL.

We present a case study for the model-based generation of test programs (i.e, the basis for a certification kit) for a realistic model of a RISC processor called VAMP. VAMP is inspired by IBM's G5 architecture. In the Verisoft project[1], a formal model for both the processor and a small operating system has been developed in Isabelle/HOL. We will adapt and reuse the processor model to generate test cases that can be used to check if a given hardware conforms to the model of the VAMP processor. The presented test scenario is of particular interest for the higher levels of certification processes as imposed by Common Criteria EAL 7. Even if the transition from C programs to the processor models has been completely covered by deductive verification methods as in CompCert [Ler09], certification bodies will require test sets checking the conformance of the underlying processor model to real hardware.

At present, specification-level verification and the development of test sets are usually two distinct tasks. Moreover, test sets for certification kits are usually developed manually. In contrast, our model-based test case generation approach uses the design model that was already used for the verification task. In particular, we are using HOL-TestGen to generate test sequences generated from the VAMP model. As HOL-TestGen is built on top of Isabelle/HOL, i. e., test specification are expressed in terms of higher-order logic (HOL), we can directly benefit from the already existing verification

[1] http://www.verisoft.de

models. In fact, the tight integration of a verification and a test environment is a distinguishing feature of HOL-TestGen.

5.2 The VAMP Model

The Verified Architecture MicroProcessor (VAMP) [BJK+06] is a 32-bit RISC CPU with a DLX-instruction set including floating point instructions, delayed program counter, address translation, and support for maskable nested precise interrupts. The VAMP hardware contains five execution units: the Fixed Point Unit, the Memory Unit, and three Floating Point Units. Instructions have up to six 32-bit source operands and produce up to four 32-bit results. The memory interface [BJK+06] of the VAMP consists of two Memory Management Units that access instruction and data caches, which in turn access a physical memory via a bus protocol.

In the context of the Verisoft project, an Isabelle/HOL specification (programmer's model) of the VAMP processor was introduced. The processor consists of a set of transitions defined over the Instruction Set Architecture (ISA) configurations. A configuration is composed of five elements:

1. *Program counter (pcp):* a 30 bit register containing the address of next instruction to be executed, this register is used to fetch an instruction without altering the execution of the current one. This pipelining mechanism is called *delayed pc*.
2. *Delayed program counter (dpc):* a 30 bit register for delayed program counter, containing the currently executed instruction. While the fetch of the next instruction is performed in the *pcp* register, the *dcp* is kept unchanged until the end of the execution of the current instruction.
3. *General purpose registers (gprs):* a register file consisting of 32 registers of 32 bits each. These registers are used in different operations, and can be addressed by their index (0–31). The first register is always set to 0.
4. *Special purpose registers (sprs):* a register file consisting of 32 registers of 32 bits each, used for particular tasks. The first register for instance is the *status* register, containing the interrupts masks. Some registers are used as flags registers or as condition registers. Each special purpose register is addressed directly by its name.
5. *Memory model (mm):* a 2^{32} bytes addressable memory. Different caching and virtual memory infrastructures are implemented in the VAMP system.

The transition relation is defined by the execution of the program instructions defined in the initial configuration. The VAMP implements the full DLX instruction set from [HP06]. This set includes load and store operations for double words, words, half words and bytes. It includes also different

shift operations, jump-and-link operations and various arithmetic and logical operations.

To avoid the complex and inconvenient bit vector representation of data and instructions, an assembly language was introduced abstracting the VAMP ISA. In this case addresses are represented by natural numbers and registers and memory contents by integers. Our test specifications and experiments are based on this instruction set (assembler) model.

The Isabelle theory of the assembler model is an abstraction of the instruction set architecture. In addition to the representation of addresses as naturals and values as integers, some other ISA features are abstracted. The instructions are represented in an abstract datatype with *readable* names. The address translation is not visible at this level, assembler computations live in linear (virtual) memory space. Interrupts are not visible at this level as well. The assembler configuration is an abstraction of the ISA configuration, defined as a record type with the following fields:

- *pcp:* a natural number representing the program counter,
- *dcp:* a natural number representing the delayed program counter,
- *gprs:* a list of integers representing the general purpose register file,
- *sprs:* a list of integers representing the special purpose register file,
- *mm:* a memory model represented by a mapping from naturals to integers.

The HOL definition of the configuration is given by the $\texttt{ASMcore}_t$ record type. The register file type is defined as a list of integers representing the different registers.

```
1  type_synonym regcont   = int -- {* contents of register *}
2  type_synonym registers = regcont list -- {* register file *}
3  type_synonym mem_t = nat ⇒mem_cell_t -- {*memory *}
4
5  record ASMcore_t = dpc :: nat
6                     pcp  :: nat
7                     gprs :: registers
8                     sprs :: registers
9                     mm   :: mem_t
```

Since the assembler representation of addresses and values is less restrictive than the bit vector representation, some conversion functions and restriction predicates were defined to reduce the domain of addresses and values to only meaningful values. This was the case also for the configurations, since the number of registers is not mentioned in the definition of the registers type. The *well-fomedness* of assembler configurations is given by the `is_ASMcore` predicate. This predicate ensures that register files contain exactly 32 registers each. It also checks that all register and memory cells contain valid values.

```
definition is_ASMcore :: ASMcore_t ⇒bool where
 is_ASMcore st ≡ asm_nat (dpc st) ∧
                  asm_nat (pcp st) ∧
                  length (gprs st) = 32 ∧
                  length (sprs st) = 32 ∧
                  (∀ ind < 32. asm_int (reg (gprs st) ind)) ∧
                  (∀ ind < 32. asm_int (sreg (sprs st) ind)) ∧
                  (∀ ad. asm_int (data_mem_read (mm st) ad))
```

The instruction set of the assembler is defined as an abstract datatype `instr` in Isabelle. All operations mnemonics are used as datatype constructors, associated to their corresponding operands. Different types of instructions can be distinguished: data transfer commands, arithmetic and logical operations, test operations, shift operations, control operations and some basic interrupts.

```
datatype instr =
  -- {* data transfer (memory) *}
    Ilb regname regname immed
  |...
  -- {* data transfer (constant) *}
  | ...
  -- {* data transfer (registers) *}
  | Imovs2i regname regname
  | ...
  -- {* arithmetic / logical operations *}
  | Iaddio regname regname immed
  | Iaddi regname regname immed
  | Iaddo regname regname regname
  -- {* test operations *}
  | Iclri regname
  | ...
  -- {* shift operations *}
  | Islli regname regname shift_amount
  | ...
  -- {* control operations *}
  | Ibeqz regname immed
  | ...
  -- {* interrupt *}
  | Itrap immed
  | ...
```

An inductive function is defined over the assembler instructions to provide the semantics of each operation. This function returns for each configuration

and instruction, the configuration resulting from executing the instruction in the initial configuration.

```
fun exec_instr :: [ASMcore_t, instr] ⇒ASMcore_t
where
  -- {* Arithmetic Instructions *}
  exec_instr st (Iaddo RD RS1 RS2) =
      arith_exec st int_add (reg (gprs st) RS1)
                                    (reg (gprs st) RS2) RD
 | ...
  -- {* Logical Instructions *}
 | exec_instr st (Iand RD RS1 RS2) =
      arith_exec st s_and (reg (gprs st) RS1)
                                    (reg (gprs st) RS2) RD
 | ...
  -- {* Shift Instructions *}
 | exec_instr st (Isll RD RS1 RS2) =
      arith_exec st sllog (reg (gprs st) RS1)
                                    (reg (gprs st) RS2) RD
 | ...
```

The transition relation is defined as a function that takes a configuration and returns its successor. The transitions are defined by the execution of the current program instruction given in the delayed program counter.

```
definition Step :: ASMcore_t ⇒  ASMcore_t
where     Step st ≡    exec_instr st (current_instr st)
```

These transition relations are used in our study as the basis of test specifications. The assembler model is more abstract than the processor model, consequently, different complex details are made transparent. Examples are interrupts handling and virtual memory and caching, pipelining and instruction reordering. In a black-box testing scenario, an abstract description of the system under test is used as a basis for test generation. This will be the case in our study, where the processor model is used to extract abstract test cases for the processor. The aim of this testing scenario is to check that the processor behaves as described in the assembler model, independently of the internal implementation details.

5.3 Testing VAMP Processor Conformance

As motivated earlier, we will apply essentially two testing scenarios: model-based *unit testing* and *sequence testing*. In a unit testing scenario, the test specification is described by pre- and post-conditions on the inputs and results produced by the system under test. This scenario assumes control over

the initial state and the access to the internal states of the SUT after the test. In sequence testing scenario, only the control of the internal state initialization is necessary, and in some cases the reference to the final state. In principle, the test result is inferred from a sequence system inputs and observed outputs. For any given inputs and state, the system—defined as an i/o stepping function—may either fail or produce outputs and a successor state. The unit testing scenario can be seen as a special form of (one step) sequence testing, where the output state is more or less completely accessible for the test.

In our case study, both testing scenarios are useful. The unit testing scenario will be used to test individually each operation or instruction with different data. Sequence testing will be used to test any sequence of instructions up to a given length. We will address subsets of related instructions separately, a combination of different instruction types is possible but not explored here. We studied four types of instructions: 1. memory related load and store operations, 2. arithmetic operations, 3. logic operations and 4. control-flow related operations.

5.3.1 Generalities on Model-based Tests

A general test specification for unit instruction testing would be the following:

```
1  test_spec pre σ ι ⟹ SUT σ ι =_k exec_instr σ ι
```

where _ =$_k$ _ is a specially defined executable equality that compares the content of the registers and just the top k memory cells (instead of infinite memory). _ =$_k$ _ is our standard conformance relation comparing the state controlled according to the model and the state controlled by the `SUT`; here, we make the testability assumption that we can trust our test environment that reads the external state and converts it to its abstraction. Note that `SUT` is a free variable that is replaced during the test execution with the system under test.

Each test case is composed of an instruction, an initial configuration and the resulting configuration after the execution of the instruction. From this test specification, HOL-TestGen will produce tests for all possible instructions.Subsets of instructions are isolated by adding a pre-condition in the test specification, specifying the type of the instruction.

For instruction sequence testing, based on the combinators from the state-exception monad (see section 4.2) `mbind`, bind _ ←_; _ and the assertion `assert_SE` a test specification can be given specifications of *valid test sequences* from initial state σ_0. In general, there are two kinds of sequence test scenarios: those who involve just observations of the executions of the local steps and those who involve a test over the final state. The former class

is irrelevant in our application domain since the local steps are just actions not reporting a computation result. However, the latter scenario may just involve a conformance on the entire state:

```
test_spec
pre  ιs::instr list ⟹
 (σ 0 ⊨(_ ←mbind ιs execVAMP; assert_SE (λσ. σ    =k SUT σ0 ιs)))
```

or just a bit of it, e. g., where a computation is finally loaded into register `0` which is finally compared:

```
test_spec
pre ιs::instr list ⟹
 (σ 0 ⊨(_ ←mbind (ιs@[load x 0]) execVAMP;
        assert_SE (λσ. (gprs σ)!0 = (gprs (SUT σ0ιs))! 0)))
```

which requires that the last load action(s) are tested before, but makes less assumptions over the execution environment (i. e., a trustworthy implementation of `_ =k _`). In both schemes σ_0 is the initial state and `ιs` is the sequence of instructions that will be generated and $\texttt{exec}_{\texttt{VAMP}}$ is a lifting of `exec_instr` into the state exception monad:

```
definition execVAMP
where execVAMP ≡ (λ i σ. Some ((), exec_instr σ i))
```

The pre-conditions `pre` of our test specifications—also called test purposes—are added to the test specifications to reduce the generated instruction sequences to any given subset.

The initial configuration can also be generated as an input of the test cases. This may produce ill-formed configurations due to their abstract representation in the assembler model. We choose for our study to define and use an empty initial configuration σ_0 that is proved to be well-formed.

5.3.2 Test Specification

Common analysis techniques such as stuck-at-faults [Hay84] are based on the idea that a given circuit design—thus, an implementation—is modified by mutators capturing a particular fabrication fault model, e. g.: one or n wires connecting gates in the circuit are broken. This can be seen conceptually as a white-box mutation technique and has, consequently, all advantages and all draw-backs of an implementation-based testing method compared to all draw-backs and all advantages to its specification-based counterparts. Stack-at-faults are very effective for medium-size circuits and use the structure of the given design to construct equivalence classes tests incorporating directly

a fault model. This type of testing technique, however, will not reveal design flaws such as a write-read error under the influence of byte-alignments in the memory.

While we have a VAMP gate-level model in our hands and could have opted for testing technique on this layer, for this thesis, we opted to stay on the design level of the VAMP machine. This does not mean that we can not refine with little effort the equivalence classes underlying our tests further: instead of assuming in our test hypothesis that "one write-read of a memory cell successful, thus all write-reads in this cell successful," one could force HOL-TestGen to generate finer test classes, by exploring the byte-or the bit-level representations of registers and memory cells.

5.3.3 Testing Load-Store Operations

To formalize a test purpose restricting our first test scenario to load and store operations, the test purpose `is_load_store` is used. This predicate returns for each instruction, if it is a load/store operation or not. It is defined just as a constraint over the syntax of the VAMP assembly language:

```
abbreviation is_load_store_byte' :: instr ⇒bool
where is_load_store_byte' iw ≡
      (∃ rd rs imm.
       (is_register rd ∧is_register rs ∧is_immediate imm) ∧
        iw ∈{Ilb rd rs imm, Ilbu rd rs imm, Isb rd rs imm})

definition is_load_store :: instr ⇒bool
where     is_load_store iw ≡
            is_load_store_word' iw ∨
            is_load_store_hword' iw ∨
            is_load_store_byte' iw
```

(the analogous test cases for `is_load_store_word'` and `is_load_store_hword' iw` are ommited here for space reasons).

Introducing this predicate in the pre-condition of the test specification reduces the domain of the generated tests to load/store operations. The resulting test specification formally stating the test goal for unit test scenario is given by the following:

```
test_spec is_load_store ι⟹SUT σ0  ι  =k exec_instr σ0  ι
apply (gen_test_cases 0 1 SUT)
store_test_thm load_store_instr
```

The test case generation procedure defined in HOL-TestGen is used to preform an exhaustive case splitting on the instructions datatype. Symbolic operands are generated for each instruction to give a set of symbolic test cases. The test generation produced 8 symbolic test cases, corresponding to the different load and store operations. A uniformity hypothesis is stated on each symbolic test case, which will allow us to select one concrete *witness* for each symbolic test case. The final generation state contains 8 *schematic* test cases, associated to 8 uniformity hypotheses. The conjunction of the test cases and the uniformity hypotheses is called a test theorem.

An example of a generated test case and its associated uniformity hypothesis is given in the following. The variables starting with `??X` (e. g., `??X4`,`??X5`,) are schematic variables representing one possible witness value.

```
1  1. SUT σ0 (Ilb ??X7 ??X6 ??X5)
2       (...)
3  2. THYP ((∃x xa xb. SUT σ0(Ilb xb xa x) (...)) ⟶
4           (∀x xa xb. SUT σ0(Ilb xb xa x) (...)))
```

The second phase of test generation is the test data instantiation. This is done using the `gen_test_data` command of HOL-TestGen. One possible resulting test case is given by the following:

```
1  SUT σ0 (Ilb 1 0 1) σ_1
```

where `σ_1` is the expected final state after executing the given operation. With this kind of test cases, each operation is tested individually, in a unit test style. This kind of test will reveal design faults i. e.if the result of the operation is not correct. It also detects any undesired state modification, like changing some flags or registers.

In a similar way, load and store instruction sequences are characterized using the same predicate `is_load_store` which is generalized to entire input sequences to the combinator `list_all` from the HOL-library. Rather than using a fairly difficult to execute characterization in form of an automaton or an extended finite state-machine that introduce some form of symbolic trace, we use monadic combinators of the state-exception monad directly to define valid test sequences constrained by suitable test purposes.

```
1  test_spec
2  list_all is_load_store (ιs::instr list) ⟹
3   (σ0 ⊨(s ←mbind ιs exec_VAMP; assert_SE (λσ. σ =k SUT σ0   ιs)))
4  apply (gen_test_cases SUT)
5  store_test_thm load_stre_instr_seq
```

Note that step two is just the call to the automatic test case generation method (declaring the free variable `SUT` as the system under test of this test case), and while the third command binds the results of this step to a data-structure called *test environment* with the name `load_store_instr_seq`. The experimental evaluation of this scenario is discussed in the next section.

One possible generated test case of length 3 is given by the following subgoal:

```
1.σ0 ⊨(s ←mbind [Isw ??X597 ??X586 ??X575,
                 Ilbu ??X557 ??X546 ??X535,
                                    Ilbu ??X517 ??X506 ??X595] exec_VAMP;
       assert_SE (λσ. σ =k SUT σ0 [Isw ??X597 ??X586 ??X575,
                                   Ilbu ??X557 ??X546 ??X535,
                                   Ilbu ??X517 ??X506 ??X595]))
2. THYP ((∃x1 x2 x3 x4 x5 x6 x7 x8 x9.
   σ0 ⊨(s ←mbind [Isw x1 x2 x3, Ilbu x4 x5 x6,
                  Ilbu x7 x8 x9] exec_VAMP; (...))) ⟶
   (∀x1 x2 x3 x4 x5 x6 x7 x8 x9.
    σ0 ⊨(s ←mbind [Isw x1 x2 x3, Ilbu x4 x5 x6,
                   Ilbu x7 x8 x9] exec_VAMP; (...))))
```

where the first subgoal gives the schematic test case, and the second subgoal states the uniformity hypothesis for this case. The generation of test data is done similarly using the `gen_test_data` command, which instantiate the schematic variables with concrete values.

```
σ0 ⊨(s ←mbind [Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8] exec_VAMP;
assert_SE (λσ. σ   =k SUT σ0 [Isw 0 1 8, Ilbu 1 0 -3, Ilbu 3 2 8]))
```

this corresponds to the following assembly code sequence:

```
ISW   0 1   8
LLBU  1 0  -3
LLBU  3 2   8
```

This test programs will eventually reveal errors related to read and write sequences. Even if each operation is realized in a correct way, the sequencing may contain errors, like errors due to byte alignment or information loss due to pipelining.

In this testing scenario, we consider test post-conditions expressed on the final state of the automaton. This post-condition is expressed using the state-exception primitive `assert_SE`. This scenario is not very realistic in hardware processors, because the final state, in particular the internal processor registers, will not be directly observable. An alternative scenario

would be to consider the state-exception primitive `return` that introduces a step by step checking of the output values. This output value might be, e. g., retrieved from the updated memory cell. Test specification for this kind of scenarios is as follows:

```
test_spec
list_all is_load_store ιs ⟹
 (σ 0 ⊨(s ←mbind ιexec_VAMP'; return (SUT ιs)))
```

which require a modified VAMP where individual steps were wrapped into trusted code that makes, e. g., internal register content explicit.

5.3.4 Testing Arithmetic Operations

Similarly, we set up a unit test scenario, where we constrain by the test purpose `is_arith` the operations to be tested to arithmetic ones:

```
test_spec σ = exec_instr σ0 i ⟹ is_arith i ⟹ SUT σ0 i σ
apply (gen_test_cases 0 1 SUT)
store_test_thm arith_instr
```

At this stage, each arithmetic operation is covered by one generated test case, an example is given in the following:

```
1. SUT σ0 (Iaddi ??X277 ??X266 ??X255) (...)
```

which contains a test case for he addition operation.

A note on the test granularity is at place here: as such, the granularity that HOL-TestGen applies to test arithmetic operations is fairly coarse: just one value satisfying all constraints over a variable of type integer is selected. This is a consequence of our model (registers were represented as integers and not as bitvectors of type: `32 word` which would be (nowadays) a valuable alternative) as well as the HOL-TestGen heuristics to select for each variable just one candidate. The standard workaround would be to introduce in the test purpose definitions more case distinctions, e. g., by `x ∈{MinInt} ∪{-50 .. -100} ∪{0} ∪ {50 .. 100}` which result in finer constraints for each of which a solution in the test selection must be found.

The sequence scenario is analogously:

```
test_spec
list_all is_arith (ι::instr list) ⟹
 (σ 0 ⊨(s ←mbind ιs exec_VAMP; assert_SE (λσ. σ=k SUT σ0 ι)))
apply (gen_test_cases SUT)
store_test_thm arith_instr_seq
```

A possible generated sequence is given in the following, resulting from the gen_test_data command.

```
1  σ 0 ⊨(s ←mbind [Isub 2 1 0, Iadd 1 5 2, Iadd 1 0 4] exec_VAMP;
2  assert_SE (λσ. σ =_k SUT σ_0 [Isub 2 1 0, Iadd 1 5 2, Iadd 1 0 4]))
```

which corresponds to the following assembly code sequence:

```
ISUB 2 1 0
IADD 1 5 2
IADD 1 0 4
```

This sequence corresponds to a subtraction followed by two addition operations.

5.3.5 Testing Control-Flow Related Operations

Also with branching operations we are following the same theme:

```
1  test_spec is_branch i ⟹SUT σ_0 i =_k exec_instr σ_0 i
2  apply (gen_test_cases 0 1 SUT)
3  store_test_thm branch_instr
```

This generates unit test cases for branching operations starting from the initial sate σ_0. One example of the generated schematic test cases is given by:

```
1  1. SUT σ_0 (Ijalr ??X27X7) (...)
```

The problem with this scenario is that the initial state is fixed, while the branching operations behavior depends essentially on the flag values. A more interesting scenario would be to consider different initial states, where the flags values are changed for each test case.

In the test sequence generation, the test specification is given as follows:

```
1  test_spec
2  list_all is_branch (ιs::instr list) ⟹
3   (σ 0 ⊨(s ←mbind ιs exec_VAMP; assert_SE (λσ. σ=_k SUT σ_0 ιs)))
4  apply (gen_test_cases SUT)
5  store_test_thm branch_instr_seq
```

The test sequence and test data generation returns, e. g., this concrete test sequence:

```
1  σ 0 ⊨(s ←mbind [Ij 1, Ijalr 0] exec_VAMP;
2           assert_SE (λσ. σ =_k SUT σ_0 [Ij 1, Ijalr 0]))
```

which corresponds to the following assembly code sequence:

```
IJ     1
IJALR  0
```

The test data generation in all the considered scenarios is performed by constraint solving and random instantiation. This leads to test sequences with coarsely grained memory access. As such, an underlying fault-model is somewhat arcane (i. e., interferences of operations in distant memory areas). If one is interested in such faults, a more dense test method should be chosen.

Rather, one would adding additional constraints to reduce the uniformity domain again. One could simply bound the range of addresses to be used in test sequences, or define a used-predicate over input sequences that computes the set of addresses that store-operations write to, and constrain the load-operations to this set, or the like. This kind of constraints can also be used to improve the coverage of our selected data, by dividing the uniformity domain into different interesting sub-domains.

5.4 Experiences and First Experimental Data

Methodologically, we deliberately refrained in this paper to modify the model—we took it "as is," and added derived rules to make it executable in test scenarios where we assume a reference implementation running against the SUT. For example, the model describes padding functions for bytes, words, and long-words treating the most significant bit differently in certain load and store operations; in the semantic machine model as it was developed in the Verisoft Project, there are comparisons on these padding *functions* themselves—this is possible in HOL, but in no functional executable language, had therefore to be replaced by equivalent formulations exploiting the fact there are only three variants of padding functions, thus a finite number, were actually *used* in the VAMP machine. Another issue is the linear memory in the machine (a total, infinite function from natural numbers to memory cells, i. e., long words); comparisons on memory, as arising in tests where the real state has to be compared against the specified state, had to be weakened to finitized conformance relations.

While as a whole, our approach is done in a pretty generic model-based testing framework, a few adaptions had to be made due to some specialties of this model. For example, since the assembly language has 56 instructions, case-splitting over the language explodes fast over the length of test sequences. While sequence tests are methodologically and pragmatically more desirable (less control over the state is assumed), they are therefore more vulnerable

to state-space explosion: sequences of length 3 generate at some point of the process $56+56^2+56^3 = 178808$ cases. In this situation, a few heuristic adaptions (represented on the tactic level) and more significantly, constraints on the level of the test purposes had to be imposed with respect to state-space explosion, test purposes like `list_all is_logic` ι helps to reduce the test sequences to $7 + 7^2 + 7^3$, i. e., a perfectly manageable size (see discussion in the next section).

5.4.1 Test Generation

As mentioned earlier, we opted for a combination of unit and sequence test scenarios. Unit tests have the drawback of imposing stronger assumptions on testability: it is assumed that the test driver has actually access to registers and memory (which essentially boils down to the fact that we trust code in the test driver that consists of store-operations of registers into the memory). Sequence tests rely on the observed behavior of tests and make weaker assumptions on testability, for the price of being more vulnerable to state-space explosion.

The sequence scenarios on load-and store operations in subsection 5.3.3 uses 39 seconds in the test partitioning phase and 42 seconds in the test data selection phase (measurements were made on a Powerbook with a 2.8 Ghz Intel Core 2 Duo). 1170 subgoals were generated, where one third are explicit test hypothesis and two third are actual test cases. The other scenarios in subsection 5.3.5, subsection 5.3.4 and the more basic subsection 5.3.1 use considerably less time (between two and twenty seconds for the entire process).

5.4.2 Test Execution

Nevertheless, compile time for the model (as part of the test drivers) was less than a second; compilation of the entire test driver in SML depends, of course, drastically on the size of finally generated tests. Since we restrained via test purposes the test cases in each individual scenario to about 1000, the compile time for a test remained below 3 seconds. Scaling up our test plan is essentially playing with a number of control parameters; however this is usually done only at the end of the test plan development for reasons of convenience.

Our study focuses for the moment on *test generations*; we did not do any experiments against hardware so far. However, there is a hardware-simulator in the sources of the Verisoft-project; in the future, we plan to generate mutants of this simulator and get thus experimental data on the bug-detection capabilities on the generated test sets.

To give an idea on how the test cases will be executed, we did some exper-

iments using the generated executable model. Starting from the abstract model, an executable translation of it in SML is performed using the Isabelle's code-generation facilities. This generated code contains all the type and constant definitions that are needed to execute the different assembler operations on an executable state. A sketch of the generated SML code for the VAMP processor is given in the following:

```
structure VAMP : sig
  datatype num = One | Bit0 of num | Bit1 of num
  datatype 'a set = Set of 'a list | Coset of 'a list
  datatype instr =
    Ilb of IntInf.int * IntInf.int * IntInf.int |
    ...
    Ijr of IntInf.int | Itrap of IntInf.int | Irfe
  val int_add : IntInf.int -> IntInf.int -> IntInf.int
  val int_sub : IntInf.int -> IntInf.int -> IntInf.int
  val cell2data : IntInf.int -> IntInf.int
  val exec_instr :
      unit aSMcore_t_ext -> instr -> unit aSMcore_t_ext
  val sigma_0 : unit aSMcore_t_ext
  val execInstrs :
      unit aSMcore_t_ext -> instr list ->
      unit aSMcore_t_ext
   ...
```

where the datatype definition `instr` is generated from the instruction type definition introduced in section 5.2. the functions definitions are generated from their corresponding constants and functions defined in the model.

Our fist experiment was the application of the generated test cases on this executable model. Using the HOL-TestGen test script generation, two test scripts were generated for load/store and arithmetic operations sequence. For both cases, 585 test cases were generated and then transformed to executable testers. Running all these tests did, obviously, not reveal any error, since the same model was used for test generation and execution.

To evaluate the quality our generated test cases, we introduced some changes to the executable model, producing a mutant model. Three changes were introduced in the `int_add`, `int_sub` and `cell2data` operations of the generated SML code. In this case, a majority of tests detected the errors. For testing the arithmetic operations, we obtained:

```
Number of successful test cases: 303 of 585 (ca. 51%)
Number of warning:                 0 of 585 (ca.  0%)
Number of errors:                  0 of 585 (ca.  0%)
Number of failures:              282 of 585 (ca. 49%)
Number of fatal errors:            0 of 585 (ca.  0%)
```

For testing the load/store operations, we obtained:

```
Number of successful test cases:  54 of 585 (ca.  9%)
Number of warning:                 0 of 585 (ca.  0%)
Number of errors:                  0 of 585 (ca.  0%)
Number of failures:              531 of 585 (ca. 91%)
Number of fatal errors:            0 of 585 (ca.  0%)
```

5.5 Conlusions

5.5.1 Related Work

Formal verification is widely used in the hardware industry since at least ten years (e. g., [Fox03, SV03, BFY+97, Rus99, Har03]). Nevertheless, formal models of complete processors as well as verification approaches that provide an end-to-end verification from the application layer to the hardware design layer are rare. Besides VAMP [Dor10], notably, exceptions are [Fox03] and [AK95]. The closest related work with respect to the processor model is [Fox03] to which our approach should be directly applicable.

Similarly, test program generation approaches for microprocessor instruction sets have been known for a long time (e. g., [FT01, KKV11, MD08, SMZ05]). Among them manual approaches based on informal descriptions of the instruction set such as [FT01] or random testing approaches such as [SMZ05]. Only a few works suggest to use model-based or specification-based test program generation algorithms, e. g., [KKV11] and [MD08]. These works have in common that they are based on dedicated test models that are independently developed from the verification models. [MD08] is the most closely related work; the authors are using the explicit state model checker SMV to generate test programs from a dedicated test model for SMV that concentrates on pipelining faults. In contrast, our approach seamlessly integrates the test program generation into an existing verification tool chain, re-using existing verification models.

5.5.2 Conclusion and Future Work

We presented an approach for testing the conformance of a processor with respect to an abstract model that captures the instruction set (i. e., the assembly-level) of the processor. This abstraction level is particular important as, first, it is the level of detail that is usually available for commercial off-the-shelf (COTS) processors and, second, it is the target level of high-level compilers.

Thus, our approach can, on the one hand, support the certification of the COTS processors for which the manufacturer is neither willing to certify the

processor itself or to disclose the necessary internal details. Moreover, our approach helps to bridge the gap between the software layer (e. g., in avionics requiring certification according to DO-178 [HB07]) and the hardware layer (e. g., in avionics requiring certification according to DO-254 [HB07]).

As (embedded) systems combining hardware and software components for providing core functionality in safety critical systems (e. g., "fly-by-wire") are used more and more often, we see an increasing need for validation techniques that seamlessly bridge the gap between hardware and software. Consequently, we see this area as the utterly important one for future work: providing a test case generation methodology that can be applied end-to-end in the development process and allows for validating each development step. These test cases, called *certification kits*, are required even if compilers and processors are formally verified: The system builders require them for proving, as part of their certification process, that their are applying the tools correctly (i. e., according to their specification).

6

Testing PikeOS API

Contents

6.1 Introduction

In the following, we will outline the PikeOS model (the full-blown model developed as part of the EUROMILS project is about 20 kLOC of Isabelle/HOL code), and demonstrate how this model is embedded into our monadic testing theory.

As a foundation for our symbolic computing techniques, we refine the theory of monads to embed interleaving executions with abort, synchronization, and shared memory to a general but still optimized behavioral test framework.

This framework is instantiated by a model of PikeOS inter-process communication system-calls. Inheriting a micro-architecture going back to the L4 kernel, the system calls of the IPC-API are internally structured by atomic actions; according to a security model, these actions can fail and must produce error-codes. Thus, our tests reveal errors in the enforcement of the security model.

The chapter proceeds as follow: In section 6.2 an informal description of PikeOS IPC is presented. The section 6.3 contains the formalisation of PikeOS IPC in Isabelle/**hol!**. In order to catch the behavior of the latter a new monad combinator is introduced in subsection 6.3.4. Moreover, a generic memory model is presented in section 6.4, it is used to specify some of PikeOS IPC atomic actions, i. e. the BUF and MAP atomic actions. In order to test PikeOS IPC, our testing approach is extended by new notions, in particular these are:

- a new coverage criteria is defined in subsection 6.5.1,
- a new symbolic execution rules are derived in subsection 6.5.3,
- a new methodology for building test drivers is presented in subsection 6.5.6.

Finally, our experimental results are presented in subsection 6.5.7.

6.2 PikeOS IPC Protocol

The IPC mechanism [SYS13a, SYS13b] is the primary means of thread communication in PikeOS. Historically, its efficient implementation in L4 played a major role in the micro-kernel renaissance after the early 1990s. Micro-kernels had received a bad reputation, as systems built on top were performing poorly, culminating in the billion-dollar failure of the IBM Workplace OS.

A combination of shared memory techniques—the MMU is configured such that parts of virtual memory space are actually represented by identical parts of the physical memory—and a radical redesign of the IPC primitives in L4 resulted in an order-of-magnitude decrease in IPC cost. Also in PikeOS, IPC message transfer can operate between threads which may belong to different tasks. However, the kernel controls the scope of IPC by determining, in each instance, whether the two threads are permitted to communicate with each other. IPC transfer is based on shared memory, which requires an agreement between the sender and receiver of an IPC message. If either the sending or the receiving thread is not ready for message transfer, then the other partner must wait. Both threads can specify a timeout for the maximum time they are prepared to wait and have appropriate access-control rights. Our IPC model includes eight *atomic actions*, corresponding more-or-less to code sections in the API system calls `p4_ipc_buf_send()` and `p4_ipc_buf_recv()` protected by a global system lock. If errors in these actions occur—for example for lacking access-rights—the system call is *aborted*, which means that all atomic actions belonging to the running system call as well as the call of the communication partner were skipped and execution after the system calls on both sides is continuing as normal. It is the responsibility of the application to act appropriately on error-codes reported as a result of a call. In our sequence test scenarios, and using our symbolic execution process running on the top of HOL-TestGen, we show how we generate tests from our formal model of the IPC mechanism, we build a *test driver* and show how we can run the generated tests against the PikeOS IPC implementation defined in C-level.

6.3 PikeOS Model

We model the protocol as composition of several operational semantics; this composition is represented by monad-transformers adding, for example, to the basic transition semantics the semantics for abort behavior.

6.3.1 State

In our model, the system state is an abstraction of the VMIT (which is immutable) and mutable task specific resources. It is presented by the (polymorphic) record type:

```
record
('memory,'thread_id,'thread,'sp_th_th,'sp_th_res,'errors)kstate=
  resource              :: 'memory
  current_thread        :: 'thread_id
  thread_list           :: 'thread list
  communication_rights  :: 'sp_th_th
  access_rights         :: 'sp_th_res
  error_codes           :: 'errors
  errors_tab            ::  'thread_id ⇀ 'errors
```

Note that the syntax is very close to functional programming languages such as SML or OCaml or F#. The parameterization is motivated by the need of having different abstraction layers throughout the entire theory; thus, for example, the *resource* field will be instantiated at different places by abstract shared memory, physical memory, physical memory and devices, etc.—from the viewpoint of an operating system, devices are just another implementation of memory. In the entire theory, these different instantiations of `kstate` were linked by abstraction relations establishing formal refinements. Similarly, the field *current_thread* will be instantiated by the model of the *ID* of the thread in the execution context and more refined versions thereof. *thread_list* represents information on threads and there executions. The *communication _rights* field represent the communication policy defined between the active entities (i. e., threads and tasks). The field *access_rights* represent the access policy defined between active entities and passive entities (i. e., system resources).

For the purpose of test-case generation, we favor instances of `kstate` which are as abstract as possible and for which we derived suitable rules for fast symbolic execution.

6.3.2 Actions

As mentioned earlier, the execution of the system call can be interrupted or *aborted* at the border-line of code-segments protected by a lock. To avoid the complex representation of interruption points, we model the effect of these lock-protected code-segments as atomic actions. Thus, we will split any system call into a sequence of atomic actions (the problem of addressing these code-segments and influencing their execution order in a test is addressed in the next section). Atomic actions are specified by datatype as follows:

```
datatype ('ipc_stage,'ipc_dir)action_ipc = IPC 'ipc_stage 'ipc_dir
datatype p4_stage_ipc = PREP | WAIT | BUF | MAP | DONE
```

```
datatype ('thread_id ,'addresses) p4_direct_ipc =
                SEND 'thread_id 'thread_id 'addresses
              | RECV 'thread_id 'thread_id 'addresses

type_synonym ACTION_ipc =
(p4_stage_ipc,(nat×nat×nat,nat list)p4_direct_ipc)action_ipc
```

Where $\texttt{ACTION}_{\texttt{ipc}}$ is type abbreviation for IPC actions instantiated by $\texttt{p4\_direct}_{\texttt{ipc}}$. The type $\texttt{ACTION}_{\texttt{ipc}}$ models exactly the input events of our monadic testing framework. Thread IDs are triples of natural numbers that specify the resource partition the thread belongs to as well as the task and the individual id. The stepping function as a whole is too complex to be presented here; we limit ourselves to presenting a portion of an auxiliary function of it that models just the PREP_SEND stage of the IPC protocol; it must check if the task and thread id of the communication partner is allowed in the VMIT, if the memory is shared to this partner, if the sending thread has in fact writing permission to the shared memory, etc. The VMIT is part of the resource, so the memory configuration, and auxiliary functions like is_part_mem_th allow for extracting the relevant information from it. The semantic of the different stages is described using a total functions:

```
definition PREP_SEND ::ACTION_ipc state_id⇒ ACTION_ipc ⇒ACTION_ipc state_id
where PREP_SEND σ act =
   (case act of (IPC PREP (SEND caller partner msg)) ⇒
     ...
     if is_part_mem_th (get_thread_by_id'' partner σ) (resource σ)
     then
       if IPC_params_c1 (get_thread_by_id'' partner σ)
       then ...)
```

Where PREP_SEND, WAIT_SEND, BUF_SEND, and DO NE_SEND define an operational semantic for the atomic actions of the PikeOS IPC protocol.

6.3.3 Traces, executions and input sequences

During our experiments, we will generate *input sequences* rather than traces. An input sequence is a list of a datatype capturing atomic action input syntactically. An *execution* is the application of a transition function over a given input sequence. Using mbind, the execution over a given input sequence *is* can be immediately constructed.

```
definition execution = (λis ioprog σ. mbind is ioprog σ)
```

6.3.4 Aborted Executions

Our model support the notion of abort. An abort is an action done by the system to stop the execution of a given system call. A system call can be aborted for different reasons:

- timeouts: a system call can not finish its execution because a timeout happened. For instance, a caller tried to access to a given resource and run out of the specified waiting time without success, i. e. the resource was not available at that moment. Or the caller run out of the specified waiting time when he was about to wait for a given input from another call.
- other error codes: a system call can not finish its execution because of a returned error code during its execution, i. e. on of the call conditions was not satisfied, e. g. wrong communication partner. Thus, the system stops the execution of the call.

In all cases, when an abort happen to a given PikeOS call, the remaining atomic actions of the call are canceled (not executed). For the case of the IPC protocol both calls, the one coming from the caller and the one coming from his communication partner, are canceled. To express the behavior of the abort in our model we will add to our specification language a new monad combinator. The behavior expressed by this combinator is abstracted by the pseudo code in algorithm 2.

In the case of an aborted system call, the semantic of our combinator express the same behavior as stutter steps in automata models, i. e. we stay in the same state, only the *error table* will change. The error table is modeled by the field `errors_tab` of the record `(...) kstate` representing the system state, the field is instantiated by a partial function with type error_tab:: thid $\rightharpoonup$ error, and it is used to save (i. e. marks by a flag) the threads in *error state*, i. e. threads who cause errors during the execution of their system call. Every thread inside the error table is considered as a thread in an error state, when a given system call executed by a given thread is aborted, i. e. the executed action provide an output error code, we update the thread table by adding the thread and its error. Before executing any atomic action (stage) we will check the error table, if a given thread executing an action different from DONE is in the domain of the function that specify the error table, then we purge his executed action (we do nothing to the state of the system) else we will execute the action. During every DONE action execution, if the thread is in the error table then, we remove it from the domain of the function that specify the error table else, we execute the DONE action.

The **hol!** representation of the new monad operator is $\texttt{abort}_{\texttt{lift}}$, the latter express the explained behavior and will be wrapped around our transition function for PikeOS IPC protocol. The wrapper transforms the behavior of

```
if executing DONE stage then
    if an error happened then
        Update error table by removing the error flag of the current
         thread and don't execute the DONE action.
    else
        Execute the DONE action.
    end
else
    if Executing a different IPC stage from DONE then
        if an error happened then
            Update the error table by putting an error flag on both
             threads in the IPC communication, the caller and his
             partner, and purge the executed action.
        else
            Execute the action.
        end
    end
end
```

Algorithm 2: A pseudo code for the Abort operator

the basic transition function related to IPC protocol presented in subsection 6.3.5, to a the behavior abstracted by algorithm 2.

```
fun abort_lift ::
    (ACTION_ipc ⇒(errors, (ACTION_ipc,'a) state_id_scheme)Mon_SE) ⇒
     (ACTION_ipc ⇒(errors, (ACTION_ipc,'a) state_id_scheme)Mon_SE)
where abort_lift ioprog a σ=
 (case a of
   (IPC DONE (SEND caller partner msg)) ⇒
     if caller ∈dom (act_info (th_flag σ))
     then unit_SE (fst (the((act_info (th_flag σ)) caller)))
                  (*shoud be: my error*)
               (σ(|th_flag := (th_flag σ)
                           (|act_info := ((act_info (th_flag σ))
                                          (caller := None))|)|))
     else unit_SE (NO_ERRORS) (σ) (*execute done*)
    (...)
   | (IPC _ (SEND caller partner msg)) ⇒
      if caller ∈dom (act_info (th_flag σ))
      then unit_SE(get_caller_error caller σ(*should be: my error*)) σ
           (* purge *)
      else (case ioprog a σof
             None ⇒None (*never happens in our exec fun*)
           | Some(NO_ERRORS, σ') ⇒unit_SE(NO_ERRORS) (σ')
           | Some(out', σ') ⇒unit_SE(out')
               (set_caller_partner_error caller partner σσ' out' a))
     (*both caller and partner were 'informed' to be in error-state.*)
  ( ...))
```

In subsection 6.5.3 we derive generic symbolic execution rules related to a given monad `ioprog` that specify an input output program, with the abort operator wrapped around,i. e. $\texttt{abort}_{\texttt{lift}}(\texttt{ioprog})$. We refine these rules for the specific case when the operational semantics,i. e. represented by the free variable `ioprog`, is related to PikeOS IPC.

6.3.5 IPC Execution Function

To combine the different semantics of IPC atomic actions we can use:

- The Isabelle specification construct *fun*: Express the semantic with explicit case matching on actions type in a single function.

The transition function is a total function of the form:

```
    fun exec_action ::ACTION_ipc state_id⇒ ACTION_ipc ⇒ACTION_ipc state_id
    where
      PREP_SEND_run:
      exec_action σ (IPC PREP (SEND caller partner msg)) =
       PREP_SEND σ (IPC PREP (SEND caller partner msg))|
      (...)
```

The function exec_action is adapted to the monads using the following definition:

```
definition exec_action_Mon
where    exec_action_Mon =
  (λact σ. Some (error_codes(exec_action σact),
   exec_action σ act))
```

The latter function represent the basic operational semantic for PikeOS IPC and it will be combined with the semantic of the abort operator presented in subsection 6.3.4. For instance we wrap around the function exec_action_Mon, the operator $\texttt{abort}_{\texttt{lift}}$ in order to get, $\texttt{abort}_{\texttt{lift}}$(exec_action_Mon act σ).

6.3.6 System Calls

As mentioned earlier, PikeOS system calls are seen as sequence of atomic actions that respect a given ordering. Actually, each system call can perform a set of *operations*. On system-level, the execution of some operations can be ignored by specifying the corresponding parameters in the call by null. PikeOS IPC API provides seven different calls, the most general one is the call *P4_ ipc()*. Using *P4_ ipc()*, five operations can be performed:

1. Send a copied message,
2. Receive a copied message,
3. Receive an event (not modeled),
4. Send a mapped message, and
5. Receive a mapped message.

The corresponding Isabelle model for the call is:

```
datatype ('thread_id, 'msg) P4_IPC_call =
   P4_IPC_call     'thread_id 'thread_id 'msg
 | P4_IPC_BUF_call 'thread_id 'thread_id 'msg
 | P4_IPC_MAP_call 'thread_id 'thread_id 'msg
   (...)
```

6.4 A Generic Shared Memory Model

Shared memory is the key for the L4-like IPC implementations: while the MMU is usually configured to provide a separation of memory spaces for different tasks (a separation that does not exist on the level of physical memory with its physical memory pages, page tables, ...), there is an important exception: physical pages may be attributed to two different tasks allowing to transfer memory content directly from one task to another.

In order to model a such memory implementation, we will use an abstract memory model with a sharing relation between addresses. The sharing relation is used to model the IPC map operation, which establishes that memory spaces of different tasks were actually shared, such that writes in one memory space were directly accessed in the other. Under the sharing relation, our memory operations respect two properties:

1. Read memory on shared addresses returns the same value.
2. All shared addresses has the same value after writing.

In formal methods, the latter two properties are called *invariants*. An invariant is a property preserved by a class of mathematical object when a certain updates (changes) are performed on that class. The notion of invariants will be used in our model of shared memory. In our memory model, the two listed invariants will be preserved on a tuple type consisting of a pair of two elements: a partial function and an equivalence relation. While the partial function will specify the memory, i. e. the function represent a mapping from its domain consisting of a set of addresses to its range consisting of their corresponding data, the equivalence relation determines the different equivalent classes for addresses. Actually, these equivalent classes are resulting from the different map operations performed by processes of a system. In order to implement this model on top of Isabelle/**hol!** we will use the specification construct `typedef`, and this for two reasons:

1. It offers a way to define an abstract type that can be equipped with invariants.
2. A defined operation on that abstract type, can be easily used for code generation and this, only by providing a soundness proof which express that the operation preserve the invariants on the defined type.

The **hol!** specification for our memory abstract type is done by:

```
1  typedef ('α, 'β) memory =
2  {(σ::'α ⇀ 'β, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
3   proof
4     show (Map.empty, (op =)) ∈?memory
5       by (auto simp: identity_equivp)
6   qed
```

This type definition defines an isomorphism between the set on the right hand side that contains pairs of the type `('α ⇀ 'β) × ('α ⇒ 'α ⇒ bool)` and the set defined by the new type `('α, 'β)memory` ; the first element of a pair is a

partial function representing a mapping from addresses to data, the second element is an equivalence relation. The type `('α, 'β)memory` is introduced by two fresh constant symbols, the function `Abs_memory` for abstracting the pairs, and `Rep_memory` the concretization function that refer to the pairs. The application of a given operation `op` on the pairs is isomorphically the same as the application of `Abs_op` on the type `('α, 'β)memory` with the only difference: the use of the type `('α, 'β)memory` for the definition of the different operations assure that the latter talk about representatives which preserve the invariant. Because the set of tuples of type `('a ⇀ 'b) × ('a ⇒ 'a ⇒ bool)` is infinite and may contain tuples that does not preserve the desired invariant, thus the direct use of `op` is not consistent. That is why we will always define a function on representatives in the following, and this in order to get the desired effects on the pairs. Afterwards we implement and use its corresponding abstraction that refers implicitly to representatives preserving the invariant.

Implecitely, five theorems are generated by Isabelle for the functions `Abs_memory` and `Rep_memory`, where `Rep_memory_inverse, ...` are names for the generated theorems:

```
1   Rep_memory_inverse:
2   Abs_memory (Rep_memory x) = x
3
4   Abs_memory_inverse:
5   ?y ∈{(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σ y)} ⟹
6   Rep_memory (Abs_memory ?y) = ?y
7
8   Rep_memory_inject:
9   (Rep_memory ?x = Rep_memory ?y) = (?x = ?y)
10
11  Rep_memory:
12  Rep_memory ?x ∈{(σ, R). equivp R ∧(∀x y. R x y ⟶σx = σy)}
```

These theorems will help in the proof of the different lemmas used for reasoning on a defined constant based on the type `('α, 'β)memory`. Using this new defined abstract type we will now specify three main memory operations, which are *write* denoted by `_ :=$ _` *read* by `_ $ _` and *map* by `_ (_ ⋈_)`. The **hol!** specification of these memory operations is represented for instance, for the case of the map operation by:

```
fun transfer_rep ::('a ⇀'b) ×('a ⇒'a ⇒bool) ⇒'a ⇒'a ⇒
                   ('a ⇀ 'b) ×('a ⇒ 'a ⇒ bool)
 where transfer_rep (m, r) src dst =
 (m o (id (dst := src)),
 (λ x y . r ((id (dst := src)) x) ((id (dst := src)) y)))
```

```
lift_definition
add_e :: ('a,'b)memory ⇒'a ⇒'a ⇒('a, 'b)memory (_ '(_ ⋈_'))
  is transfer_rep using transfer_rep_sound
  by simp
```

The function `transfer_rep` is an update function on representatives, i. e. on the pairs of type `('a ⇀ 'b) × ('a ⇒ 'a ⇒ bool)`, and the function `add_e` is its abstraction defined on the type `('α, 'β)memory`.

Basically, the function `transfer_rep` takes a memory represented by the pair `('a ⇀ 'b) × ('a ⇒ 'a ⇒ bool)`, a source address `src`, a destination adress `dst` and update the pair, in order to express the effect of a memory map on that pair, as follow:

1. the first element of the pair, which is a partial function representing a mapping from addresses to data, is updated by assigning the data of the source address to the destination address

2. the second element of the pair, which is an equivalent relation between addresses, is updated by adding the destination address to the same equivalent class of the source address, and at the same time the relation between the destination and its old equivalent class is destroyed. This definition was validated by PikOS kernel engineers

Actually, we will not directly use `transfer_rep`, the function will be abstracted by `add_e`, and this is advantageous for the following reasons; on one hand we make sure that, on model level, `add_e` will always return pairs that preserve the invariant. On the other hand, the specification constraint `lift_definition` provide automatically a code generation setup for memory operations based on the type `('α, 'β)memory`, i. e. the generated implementation will contain implicitly only pairs that preserve the invariant.

If we look closely, we can observe that a little proof was mandatory to get the definition of `add_e`. In fact, in order to preserve the consistency of its global context, Isabelle forces a such proof. This proof is used to make sure that the invariant defined in the abstract type is preserved by the definition of `add_e`. In other words, we have to make sure that the added definition is

sound and its use does not break the invariant, a such soundness proof was provided by the following lemma:

```
lemma transfer_rep_sound:
assumes σ ∈ {(σ , R). equivp R ∧(∀x y. R x y ⟶σx = σ y)}
shows transfer_rep σ src dst ∈
      {(σ , R). equivp R ∧(∀x y. R x y ⟶σx = σ y)}
proof -
  obtain mem and R
    where P: (mem, R) = σand
          E: equivp R and
          M: ∀x y . R x y ⟶mem x = mem y
      using assms equivpE by auto
  obtain mem' and R'
    where P': (mem', R') = transfer_rep σsrc dst
      by (metis surj_pair)
  have D1: mem' = (mem o (id (dst := src)))
   and D2: R' = (λ x y . R ((id (dst := src)) x)
                ((id (dst := src)) y))
    using P P' by auto
  have equivp R'
    using E unfolding D2 equivp_def by metis
  moreover have ∀y z . R' y z ⟶mem' y = mem' z
    using M unfolding D1 D2 by auto
  ultimately show ?thesis
    using P' by auto
qed
```

In order to simplify the use of these abstract memory operations by constraint solvers, and also in order to simplify the proof of symbolic execution rules related to these operations, lemmas expressing the key properties of our shared memory model were introduced, we will present only the most important lemmas:

```
definition sharing ::
          'a ⇒ ('a, 'b)memory ⇒'a ⇒bool ((_ shares()_/ _)
where (x shares (σ) y) ≡(snd(Rep_memory σ) x y)
```

```
definition Domain :: ('α, 'β)memory ⇒'α set
where    Domain σ = dom (fst (Rep_memory σ))
```

```
lemma shares_result:
  assumes  1: (x shares (σ) y)
  shows      σ $ x = σ $ y
  using assms lookup_def shares_result
  by metis
```

Sharing is modulo equivalence relation:

```
lemma sharing_refl [simp]: (x shares (σ) x)
 using insert Rep_memory[of σ]
 by (auto simp: sharing_def elim: equivp_reflp)
```

```
lemma sharing_sym [sym]:
  assumes x shares (σ) y
  shows  y shares (σ) x
  using assms Rep_memory[of σ]
  by (auto simp: sharing_def elim: equivp_symp)
```

```
lemma sharing_trans [trans]:
   assumes x shares (σ) y
   and    y shares (σ) z
   shows  x shares (σ) z
   using assms insert Rep_memory[of σ]
   by(auto simp: sharing_def elim: equivp_transp)
```

Sharing relates to memory write as follows:

```
lemma sharing_upd: x shares (σ(a :=$b)) y = x shares (σ) y (*$*)
  using insert Rep_memory[of σ]
  by(auto simp: sharing_def update_def
     Abs_memory_inverse[OF update_sound])
```

```
lemma update_idem' :
  assumes 1: x shares (σ) y
  and     2: x ∈Domain σ
  and     3: σ $ x = z
  shows   σ(y:=$ z) = σ
proof -
 have * : y ∈Domain σ
          by(simp add: shares_dom[OF 1, symmetric] 2)
 have **: σ (x :=$ (σ $ y)) = σ
   using 1 2 *
   by (simp add: update_triv)
 also have (σ $ y) = σ$ x
   by (simp only: lookup_def shares_result [OF 1])
 finally show ?thesis
   using 1 2 3 sharing_sym update_triv
   by fast
qed
```

```
lemma update_share:
  assumes z shares (σ) x
  shows   σ(x :=$ a) $ z = a
  using assms
  by (simp only: update_apply if_trueₜ)
```

```
lemma update_other:
  assumes ¬(z shares (σ) x)
  shows   σ(x :=$ a) $ z = σ$ z (*$*)
  using assms
  by (simp only: update_apply if_False)
```

```
theorem update_cancel:
assumes x shares_σ x'
shows   σ(x :=$ y)(x' :=$ z) = (σ(x' :=$ z))  (*$*)
proof -
  have ** :
  ⋀R σ. equivp R ⟹R x x' ⟹
   fun_upd_equivp R (fun_upd_equivp R σx (Some y)) x' (Some z) =
   fun_upd_equivp R σx' (Some z)
   unfolding fun_upd_equivp_def
   using equivp_def
   by metis
  show ?thesis
  using ** Pair_code_eq Pair_upd_lifter.simps assms sharing_charn
       sharing_def update' update.rep_eq
  by metis
qed
```

```
theorem update_commute:
   assumes 1:¬ (x shares (σ) x')
   shows    (σ(x :=$ y))(x' :=$ z) =
             (σ(x':=$ z)(x :=$ y))
proof -
    (...)
```

Sharing relates to domain as follows:

```
lemma Domain_mono:
  assumes 1: x ∈Domain σ
  and     2: (x shares (σ) y)
  shows      y ∈Domain σ
  using 1 2 Rep_memory[of σ]
  by (auto simp add: sharing_def Domain_def )
```

```
lemma update_triv:
  assumes 1: x shares (σ) y
    and   2: y ∈Domain σ
  shows      σ (x :=$ (σ $ y)) = σ
proof -
  {
    fix z
    assume zx: z shares (σ) x
    then have zy: z shares (σ) y
      using 1 by (rule sharing_trans)
    have  F: y ∈Domain σ ⟹ x shares (σ) y ⟹
            Some (the (fst (Rep_memory σ) x)) = fst (Rep_memory σ) y
      by(auto simp: Domain_def dest: shares_result)
    have Some (the (fst (Rep_memory σ) y)) = fst (Rep_memory σ) z
      using zx and shares_result [OF zy] shares_result [OF zx]
      using F [OF 2 1]
      by simp
  } note 3 = this
  show ?thesis
    unfolding update'' lookup_def fun_upd_equivp_def
    by (simp add: 3 Rep_memory_inverse if_cong)
qed
```

```
lemma update_idem :
   assumes 1: x shares (σ) y
   and     2: x ∈Domain σ
   and     3: σ $ x = z
   shows   σ(x:= $ z) = σ
proof -
  have * : y ∈Domain σ
    by(simp add: shares_dom[OF 1, symmetric] 2)
  have σ (x := $ (σ $ y)) = σ
    using 1 2 * by (simp add: update_triv)
  also have (σ $ y) = σ$ x
    by (simp only: lookup_def shares_result [OF 1])
  also note 3
  finally show ?thesis .
qed
```

Similarly, we prove other rules for memory map and memory read which represent a memory theory modulo sharing. The defined memory operations are used actually to implement the MAP and BUF actions of PikeOS IPC. For more details on our **hol!** model, and the core theory for shared memory see section B.

6.5 Testing PikeOS IPC

6.5.1 Coverage Criteria for IPC

An IPC call defines a *communication* relation between two threads. In PikeOS, IPC communications can be symmetric, transitive but can not be reflexive (a thread can not send or receive an IPC message for himself). The transitivity or intransitivity of IPC communications depends mainly on the defined communication rights table and access rights table. In this section, we will define input sequences for ipc calls. The defined input sequences express IPC communications between threads. Other definitions, which are almost the same as the ones used for input sequences, will be used to derive the possible communications between threads after the execution of an IPC call. The IPC input sequences will be used in scenarios for testing information flow policy via IPC error codes, and also scenarios on access control policy implemented via the two tables cited before.

The definition of an input sequence of type IPC communication is based on a new coverage criterion. The criterion is based on the functional model of PikeOS IPC (see section 6.2), and also on our technique to reduce the set of interleaving if two actions can commute (see section 4.4).

- **Criterion3: IPC communications** (IPC_{comm}) *the interleaving space of input sequences gets a complete coverage iff all IPC communications of a given SUT are covered.*

IPC communications are input sequences derived under $\mathbf{IPC}_{comm}$. They have the form:

```
1  [IPC PREP (SEND th_id th_id' msg), IPC PREP (RECV th_id' th_id msg),
2   IPC WAIT (SEND th_id th_id' msg), IPC WAIT (RECV th_id' th_id msg),
3   IPC BUF (RECV th_id' th_id msg), IPC DONE (RECV th_id' th_id msg),
4   IPC DONE (SEND th_id th_id' msg)]
```

6.5.2 Test Case Generation Process

In our model, a test case generation process is applied on the test scenario to generate concrete tests. To apply a such process we will implicitly benefit from implemented tools, proofs and tactics of Isabelle. As explained in section 4.5, a test scenario is specified by a test specification which is actually a lemma. The goal is not to provide a proof for the lemma, the goal is just to normalize this HOL formula until we get a *test normal form (TNF)* [BW13], and then we generate concrete test from the TNF. In our approach, the process of test generation is composed of:

The Symbolic State.

In our model a symbolic state is the Isabelle lemma proof statement, i. e. a proof context.

The Symbolic Execution Process.

Our symbolic execution process can be seen as an exploration of the proof tree resulting from the application of symbolic execution rules to a given test specification. Symbolic execution rules are Isabelle proved lemmas. Those rules are inference rules derived from a given operational semantics. They are used to simulate the execution of a given transition function, which specify the behavior of the system under test. The application of a such rules allows for going from a symbolic state, i. e. a proof statement, to another symbolic state. In sequence test scenarios this step is applied until the input sequence is empty.

The Normalization Process.

Normalization rules are Isabelle proved lemmas. Two main goal are distinguished for the normalization process

1. First, normalization rules are used to simplify the abstract test cases generated after the application of symbolic execution rules, in order to get a proof statement containing a set of TNFs that can be easily treated by constraint-solvers.

2. Second, normalization rules are used to eliminate as much as possible *unfeasible executions* in the proof tree, i. e. proof statements that lead to true, (see subsection 6.5.4 for further explanation).

In our model, the outputs from this step are *abstract test cases*. Abstract test cases are a normalized proof goals generated from symbolic execution process. Proof goals are normalized, i. e., reduced to clauses over linear arithmetic, list, and map-theories in a format that can be treated by the subsequent constraint solver. Outputs from the normalization process are also called TNFs. In our approach, the step of normalization takes most of the generation time.

The Test Theorem.

After the normalization process we generate the test theorem. Actually HOL-TestGen provides a tactic for the generation of a test theorem of the

form:

$$\frac{C_1(a_1) \Rightarrow P(a_1,\ PUT\ a_1) \ \ldots\ C_n(a_n) \Rightarrow P(a_n,\ PUT\ a_n) \qquad THYP(H_1 \wedge \cdots \wedge H_n)}{TS}$$

The test theorem decompose each abstract test case in the local proof context generated from a test specification to 3 parts:

1. **Proof Obligations**: are the premises of a given abstract test case. e. g. in the previous formula a proof obligation is $C_i(a_i)$.
2. **Testing Hypotheses**: In addition to testing hypotheses expressed as assumptions of a given test specification, HOL-TestGen offer a way to introduce testing hypotheses, e. g. regularity and uniformity hypotheses, to a test specification. In the previous formula testing hypothesis are H_i. $THYP$ is a constant definition used as markup for the testing hypothesis during the generation of the test theorem.
3. **Abstract Test Cases**: also called TNFs, they are represented in the test theorem by $C_i(a_1) \Rightarrow P(a_i,\ PUT\ a_i)$, where P is the oracle, and a_i is a concrete instance that must satisfy the constraint C_i.

A test theorem state that a concrete test case passes if the application of a program under test PUT on a concrete instance a_i satisfies the oracle P.

Test Data Generation.

The proof obligations of each abstract test case are sent to constraint-solvers such as Z3[dMB08], in order to construct a concrete ("ground") data for the variables. These instantiated abstract test cases represent actually execution paths in a program under test; they are used as test cases for this system.

6.5.3 Symbolic Execution Rules

Symbolic execution rules are inference rules for the elimination of the inputs in the test specification. In our model we distinguish two categories of these rules:

1. The generic ones: they are related to operators of our specification language, i. e. the proposed monad operators in our theory like:`bind_SE`. These rules are fixed element in the theory, and they represent the generic simulation for the behavior of any state exception monad `ioprog`, of type `('ι ⇒ ('o,'σ) MON_SE)`.
2. The specific ones: they are a refinement, or an instantiation, of the generic ones. These rules represent the simulation of the behavior of a `model`, which is an intantiation of `ioprog` by a given operational sematic.

The Generic Rules.

Generic rules are elimination rules derived for the generic operational semantics expressed by the different monads operators introduced by our specification language. This kind of rules has the form:

$$\frac{(\sigma \models outs \leftarrow ioprog\ (\iota \# \iota s); P\ s) \qquad \begin{array}{c} \left[\begin{array}{l} ioprog\ \iota\ \sigma\ =\ Some\ (o_\iota, \sigma') \\ (\sigma' \models outs \leftarrow ioprog\ \iota s; P\ (o_\iota \# s)) \end{array}\right]_{o_\iota, \sigma'} \\ \vdots \\ Q \end{array}}{Q}$$

where σ is a symbolic variable that denote the state of a given system, *outs* is a sequence of outputs resulting from the execution of the transition function *ioprog*, $\iota \# \iota s$ is a list of inputs and P is a post condition on the sequence of outputs. A concrete example of generic symbolic execution rules is the rule 1 presented in section 4.2. In order to catch the behavior of PikeOS, our specification language was extended by a new state exception monad operator called $\texttt{abort}_{\texttt{lift}}$, an example of a generic symbolic execution rule related to this operator is:

```
lemma abort_wait_send_mbindFSave_E:
  assumes valid_exec:
   (σ ⊨(outs ←(mbind ((IPC WAIT (SEND caller partner msg))#S)
                        (abort_lift ioprog));P outs))
  and  in_err_state:
   caller ∈dom (act_info (th_flag σ)) ⟹
    (σ ⊨(outs ←(mbind S (abort_lift ioprog));
          P (get_caller_error caller σ# outs))) ⟹Q
          (...)
  and not_in_err_state_Some3:
   ⋀σ' error_IPC.
    (caller ∉dom (act_info (th_flag σ))) ⟹
    ioprog (IPC WAIT (SEND caller partner msg)) σ=
Some(ERROR_IPC error_IPC, σ') ⟹
    ((set_error_ipc_waitr caller partner σσ' error_IPC msg) ⊨
    (outs ←(mbind S(abort_lift ioprog));
    P ( ERROR_IPC error_IPC# outs))) ⟹Q
  and not_in_err_state_None:
   (caller ∉dom (act_info (th_flag σ))) ⟹
    ioprog (IPC WAIT (SEND caller partner msg)) σ= None ⟹
    (σ ⊨(P [])) ⟹Q
  shows Q
proof (cases caller ∈dom (act_info (th_flag σ)))
(...)
```

In order to motivate the use of elimination rules for symbolic execution, we will explain the process of their application on a given proof context. The use of the rule `abort_wait_send_mbindFSave_E` on a given test specification `Test_Scenario` is conditioned by the existence of a given assumption in `Test_Scenario` that have the same scheme of the assumption `valid_exec` and the existence of a conclusion. For the case of a valid test specification the conclusion will have the same scheme of `valid_exec`, the only difference will be the FREE variable that represent the model, e. g. `ioprog`. Actually, it is replaced by a variable, e. g. `SUT`, that represent the system under test. Once these conditions are brought together for a given test specification `Test_Scenario` the application of the rule will be performed using the tactic `ematch_tac` (see section 4.5 for further explanations). The process of the application of rules , such as `abort_wait_send_mbindFSave_E`, on a valid representation of `Test_Scenario` is:

1. Each time the input action `(IPC WAIT (SEND caller partner msg))` is in the header of a sequence of inputs ι`s` specified in a test specification `Test_Scenario`, a matching is established between the assumption `valid_exec` and the assumption that specify a model of a tested system in `Test_Scenario`, e. g. an assumption that specify a model for a test specification `Test_Scenario` related to PikeOS can be $\sigma \models$ `(outs` $\leftarrow$`mbind` ι`s(abort`$_{\texttt{lift}}$`exec_action_Mon);return(outs = x)`. The same thing will happen for the conclusion of the rule, which is by the way a free variable `Q` that can be instantiated by any boolean formula, of course for the case of a valid test specification the scheme of the conclusion specify a valid test execution for a system under test, e. g. $\sigma \models$`(outs` $\leftarrow$`mbind` ι`s SUT;return(outs = x)`.

2. After the establishment of the ematching, the proof statement provided by `Test_Scenario` is transformed to a new proof statement. The latter will contain a set of proof goals, each goal has is a "not matched" assumption specified in the rule, e. g. if `Test_Scenario` contain only an assumption in the form of `valid_exec` then the new proof context, after the application of the rule with ematching tactic, will contain the other assumptions of the rule like `in_err_state` and `not_in_err_state_Some3`, etc.

3. We repeat the same process with different rules related to different input actions until we got an empty input sequence. The resulting proof statement will receive a normalization process in order to get abstract test cases for `Test_Scenario`.

A such process, actually based on ematching technique, has an enormous performance gain effect on symbolic execution engine of Isabelle. Because, the whole calculation process is reduced technically to a formal syntactic transformation of the proof context, instead of calculus based on substitution,

rewriting, instantiation, introduction, etc. From another side, the execution of a such process on a sequence of inputs specified in a given test specification can be easily automated by an algorithm. The algorithm basically is represented by an Isabelle tactic, the latter takes the different symbolic execution rules related to the different actions of the specified system and execute the rules on the proof context until no rules can be applied. For instance, a tactic for symbolic execution related to the actions of PikeOS IPC is:

```
val abort_ipc_mbind_TestGen_PureE21_ematch =
 (ALLGOALS o TestGen.REPEAT') (CHANGED o TRY o FIRST'
 [ematch_tac
  [@{thm abort_prep_send_HOL_elim21},
   @{thm abort_prep_recv_HOL_elim21},
   @{thm abort_wait_send_HOL_elim21},
   @{thm abort_wait_recv_HOL_elim21},
   @{thm abort_buf_send_HOL_elim21},
   @{thm abort_buf_recv_HOL_elim21},
   @{thm abort_map_send_HOL_elim2},
   @{thm abort_map_recv_HOL_elim2},
   @{thm abort_done_send_HOL_elim1'},
   @{thm abort_done_recv_HOL_elim1'}]]);
```

The tactic `abort_ipc_mbind_testGen_PureE21_ematch` is implemented on `SML` level using the different Isabelle `SML` libraries, the elements of the tactic are:

- `ALLGOALS`: a tactic combinator of type `tactic * tactic -> tactic` from the module `Tactical` of Isabelle/ML. It applies the tactic on all goals of a proof statement. A proof statement is usually called a proof context.
- `TestGen.REPEAT'`: a tactic combinator of type `(int -> tactic)-> int -> tactic`. It is an adaptation of `REPEAT_ALL_NEW`, from the module `Tactical` of Isabelle/ML for HOL-TestGen and it is used to repeat the same tactic on a given subgoal.
- `CHANGED`: a tactic combinator of type `tactic -> tactic`. Its apply the tactic on a given goal, and if it fails (i. e.the goal is not changed), an Isabelle fail error is raised.
- `TRY`: a tactic combinator of type `tactic -> tactic`. its apply the tactic on a given goal, and if it fails, it let the goal unchanged.
- `FIRST'`: a tactic combinator of type `('a -> tactic)list -> 'a -> tactic`. Tries a number of tactics, specified actually inside a list, on a given goal.

- `@{thm _}`: an antiquotation that refers to a given Isabelle theorem. Antiquotations are used as links to the object specified using Isabelle's specification constructs. The objects can be Isabelle theorems, types, theories, etc. Each object has its own type of antiquotation, e. g. in order to refer to a given Isabelle theory we use `@{theory theory_name}`, another antiquotation can be `@{context}`, it is used to refer to a given local context(proof statement) of a proof. Antiquotations are useful for many activities, e. g. they are useful in order to get formal links of the different objects in a given document generated from Isabelle theories, which helps for instance in the review of the document. Also they are useful for development, e. g. in the development automated tactics.

- `abort_prep_send_HOL_elim21`: is a symbolic execution rules related to PikeOS IPC model.

For more details on Isabelle tactic development we would refer to [Urb]. Moreover note, for more details on the proofs of symbolic execution rules related to $\texttt{abort}_{\texttt{lift}}$ see section O.

The Specific Rules.

These rules are instantiations for the generic ones by a given operational semantics. For the case of PikeOS system, its operational semantics is expressed by a transition function (presented in subsection 6.3.5) over 10 atomic actions which are:

1. PREP SEND/RECV: in this stage some checks related to PikeOS message descriptor, i. e. a file containing details about the communicating threads, are done.

2. WAIT SEND/RECV: The wait stage is mainly used for synchronisation.

3. BUF SEND/RECV : The stage BUF represent data transfer via memory copy.

4. MAP SEND/RECV : The stage MAP data transfer via memory mapping.

5. DONE SEND/RECV: The stage DONE used to finish the IPC communication between the threads.

As mentioned in the previous section and in section 4.5, the role of the symbolic execution rule is to update the proof context according to the execution semantics of the different atomic actions of the IPC protocol. An example of

a symbolic execution rule derived from the operational semantics of PikeOS IPC is:

```
lemma abort_wait_send_HOL_elim21:
  assumes
  valid_exec:
  (σ |=(outs <-(mbind ((IPC WAIT (SEND caller partner msg))#S)
                (abort_lift exec_action_id_Mon)); P outs))
  and in_err_exec:
    caller ∈dom (act_info (th_flag σ)) ⟹
     (σ |=(outs <-(mbind S(abort_lift exec_action_id_Mon));
                  P (get_caller_error caller σ# outs))) ⟹Q
  and
not_in_err_exec1:
 caller ∉dom (act_info (th_flag σ)) ⟹
   IPC_send_comm_check_st_id caller partner σ⟹
   IPC_params_c4 caller partner ⟹
   IPC_params_c5 partner σ⟹
  (σ(|current_thread := caller,
     thread_list   := update_th_waiting caller (thread_list σ),
     error_codes   := NO_ERRORS,
     th_flag       := th_flag σ|)
       |=(outs <-(mbind S(abort_lift exec_action_id_Mon));
                  P (NO_ERRORS # outs))) ⟹Q
(...)
not_in_err_exec24:
 caller ∉dom (act_info (th_flag σ)) ⟹
  IPC_send_comm_check_st_id caller partner σ⟹
  IPC_params_c4 caller partner ⟹
 ¬IPC_params_c5 partner σ⟹
  ∃th. (thread_list σ) caller = Some th ⟹
 (σ(|current_thread := caller ,
    thread_list   := update_th_current caller (thread_list σ),
    error_codes   := ERROR_IPC error_IPC_5_in_WAIT_SEND,
    th_flag       := th_flag σ
    (|act_info := act_info (th_flag σ)
    (caller ↦(ERROR_IPC error_IPC_5_in_WAIT_SEND),
     partner ↦(ERROR_IPC error_IPC_5_in_WAIT_SEND))|)|) |=
    (outs <-(mbind S(abort_lift exec_action_id_Mon));
            P (ERROR_IPC error_IPC_5_in_WAIT_SEND# outs)))⟹Q
  shows Q
```

Each assumption inside the above elimination rule express a possible *execution path* for the action appearing in the head of the executed input sequence, e. g. the latter rule is related to the wait stage represented syntaxically by `(IPC WAIT (SEND caller partner msg))`.

Other Rules.

In order to simplify the proof of the symbolic execution rules presented earlier, other rules related to the execution semantics of PikeOS were derived:

```
lemma abort_prep_send_obvious10':
 (σ ⊨(outs ←(mbind ((IPC PREP (SEND caller partner msg))#S)
                   (abort_lift exec_action_id_Mon)); P outs)) =
  ((caller ∈dom ((act_info o th_flag)σ) ⟶
  (σ ⊨(outs ←(mbind S(abort_lift exec_action_id_Mon));
              P (get_caller_error caller σ# outs)))) ∧
 (caller ∉dom ((act_info o th_flag)σ) ⟶
 (∀a b. (a = NO_ERRORS ⟶
   exec_action_id_Mon (IPC PREP (SEND caller partner msg)) σ=
   Some (NO_ERRORS, b) ⟶
   (σ(|current_thread := caller,
      thread_list   := update_th_ready caller (thread_list σ),
      error_codes   := NO_ERRORS,
      th_flag := th_flag σ|)⊨
      (outs ←(mbind S(abort_lift exec_action_id_Mon));
              P (NO_ERRORS # outs)))) ∧
  (∀error_memory. a = ERROR_MEM error_memory ⟶
    exec_action_id_Mon (IPC PREP (SEND caller partner msg)) σ=
    Some (ERROR_MEM error_memory, b) ⟶
     (σ(|current_thread := caller,
        thread_list   := update_th_current caller (thread_list σ),
        error_codes   := ERROR_MEM error_memory,
        th_flag       :=
        th_flag σ
        (|act_info := ((act_info o th_flag)σ)
        (caller ↦(ERROR_MEM error_memory),
         partner ↦(ERROR_MEM error_memory))|)|)
  (...)
```

Moreover, in order to optimize the process, some rules called behavioral refinement rules are derived:

```
lemma abort_prep_send_obvious0:
assumes not_in_err :
       caller ∉dom (act_info (th_flag σ))
and    ioprog_success:
       ioprog (IPC PREP (SEND caller partner msg)) σ=
        Some(NO_ERRORS, σ')
shows abort_lift ioprog (IPC PREP (SEND caller partner msg)) σ=
       Some(NO_ERRORS, (error_tab_transfer caller σσ'))
using assms
by simp
```

For more details on these rules we would refer to section L.

6.5.4 Abstract Test Cases

Abstract test cases are proof goals resulting from the application of symbolic execution and the normalization processes on a given test specification. Abstract test cases represent a *possible* execution path in the system under test. In our approach, having n number of abstract test cases does not necessarily mean that all n paths are *feasible*. An abstract test case is feasible if and only if there exist a model, i. e. an instatiation of the free variables by a witness, that satisfy the premises of the abstract test case. In our approach, the number of feasible test cases is always less than or equal to the number of abstract test cases resulting from symbolic execution and normalization processes. The number of feasible abstract test cases is not necessarily equal to the number of concrete tests. A concrete test is a witness used to justify that a given abstract test case is feasible. Many witnesses can exist and used for the justification. Actually, in some cases the number of witnesses can be infinite. Of course, if no witnesses can be derived for an abstract test case this means that the abstract test case is *infeasible*. Thus, in our approach we can clearly end with 0 concrete tests for a given test scenario and this can happen if the constraint-solver can not provide a model that satisfies the *proof obligations* of the formula that represent an abstract test case. The problems related to detecting feasible abstract test cases, and the elimination of infeasible ones before the test generation, is not tackled during this thesis. An example of an abstract test case is:

```
⋀z za y.
(...) =
 [e, f, g] ⟹
 (...) =
 [a, b, C] ⟹
 IPC_send_comm_check_st_id thID2 thID1 σ_1 ⟹
 IPC_params_c4 thID2 thID1 ⟹
 IPC_params_c5 thID1 σ_1 ⟹
 act_info (th_flag σ_1) thID2 = None ⟹
 ¬ IPC_buf_check_st_id thID2 thID1
     (σ_1(|current_thread := thID2,
          thread_list :=
            if thID2 ∈dom (thread_list σ_1)
            then thread_list σ_1(thID2 ↦(the ∘thread_list σ_1) thID2
                 (|th_state := WAITING|))
            else thread_list σ_1,
          error_codes := NO_ERRORS|)) ⟹
 thID1 ≠ thID2 ⟹
 act_info (th_flag σ_1) thID1 = Some y ⟹
 σ_1 ⊨
 ( outs ←mbind
           [IPC WAIT (RECV thID1 thID2 [z, za]),
            IPC WAIT (SEND thID2 thID1 [z, za]),
            IPC BUF (SEND thID2 thID1 [z, za]),
            IPC MAP (SEND thID2 thID1 [z, za]),
            IPC DONE (SEND thID2 thID1 [z, za]),
            IPC DONE (RECV thID1 thID2 [z, za])]
            PUT2; unit_SE
           (outs =
            [y, NO_ERRORS,
             ERROR_IPC error_IPC_1_in_BUF_SEND,
             ERROR_IPC error_IPC_1_in_BUF_SEND,
             ERROR_IPC error_IPC_1_in_BUF_SEND,
             ERROR_IPC error_IPC_1_in_BUF_SEND]))
```

In order to get a concrete test case we have to instantiate this abstract test case with witnesses for the variables `z, za, y`. The instantiation process is done by sending the formula that contains the conjunction of the premises, e. g. `IPC_params_c4 thID2 thID1`, to constraint-solvers via an interface provided by HOL-TestGen. In our terminology, the conjunction between the premises of an abstract test case is called Proof Obligation (PO).

Most of the time, a configuration is needed in order to help the constraint solver to reason about proof obligations. The configuration of the constraint solver is basically done by a set of Isabelle lemmas that help in the solving process of the PO. For technical reasons, the lemmas of the configuration

must be written in **hol!** language, and not in `isar` or `pure` language. For example in order to allow the constraint-solver `smt` to reason about properties related to our abstract memory model, we use the rule:

```
lemma adde_share_charn [simp, code_unfold]:
 assumes 1: ¬(i shares (σ) k')
 and     2: ¬(k shares (σ) k')
 shows  i shares(σ(i' ⋈k')) k = i shares (σ) k
 using assms fun_upd_apply id_def mem_adde_E sharing_def sharing_refl
 by metis
```

In its current form this rule will be refused by the solver `smt`. The following adaptation is needed:

```
lemma adde_share_charn_smt :
 ¬(i shares (σ) k') ∧
  ¬(k shares (σ) k') ⟶
  i shares (σ(i' ⋈k')) k = i shares (σ) k
  using adde_share_charn
  by simp
```

In our framework, and in order to feed the solver `smt` with the rule `adde_share_charn_smt` we use the command:

```
declare adde_share_charn_smt [testgen_smt_facts]
```

We have to notice that we experienced several problems related to solving a PO containing constraints around an abstract type,e. g. the type of our memory model. For example, in some cases the `smt` solver fails to provide a solution to a PO containing a constraint of the form `(i shares (σ) k)`, and this of course because we do not have yet a perfect lemmas configuration that help the solver to reason about the `shares` relation correctly.

6.5.5 Test Data For Sequence-based Test Scenarios

A test scenario is represented by a test specification and can have two main schemes: unit test scheme or sequence test scheme. The specification `TS_simple_example2` is an example of a sequence test scenario for PikeOS IPC.

```
  test_spec TS_simple_example2:
    ιs ∈ IPC_communication ⟹
    σ 1 ⊨ (outs ←mbind is(abort_lift exec_action_Mon);return(outs = x)
  ⟶σ 1 ⊨ (outs ←mbind is SUT; return(outs = x))
```

For a σ_1 definition that contains a suitable VMIT configuration, a possible generated values for ι`s` are, e. g.:

```
[IPC PREP (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC PREP (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC WAIT (RECV (0,0,1) (0,0,2) [0,4,5,8]),
 IPC WAIT (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC BUF  (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (SEND (0,0,2) (0,0,1) [0,4,5,8]),
 IPC DONE (RECV (0,0,1) (0,0,2) [0,4,5,8])]
```

The sequence is an abstraction of an IPC communication between the thread with the $ID = (0, 0, 1)$ and the thread with $ID = (0, 0, 2)$ via a message $msg = [0, 4, 5, 8]$. Natural numbers inside the message are abstractions on memory addresses. In `TS_simple_example2` the execution semantic of the input sequence is represented by our execution function *exec_ action_ Mon*. We wrapped around our execution function a monad transformer $abort_{lift}$ that express the behavior of an abort. The equality in `return(outs = x)` specify our conformance relation between SUT outputs and the model outputs. After using our symbolic execution process the `out` of this test case is:

```
[NO_ERRORS,
 NO_ERRORS,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV,
 ERROR_IPC error_IPC_1_in_WAIT_RECV]
```

The error-codes observed in the sequence is related to IPC. The error-codes was returned in the stage *WAIT_ RECV*. The interpretation of this error-codes is that the thread has not the rights to communicate with his partner. We can observe the behavior of our abort operator in this sequence of error-codes; All stages following `WAIT_RECV` are purged (not executed), and the same error is returned instead. We focus on error-codes in our scenarios, since error-codes represent a potential for undesired information flow: for example, un-masked error-messages may reveal the structure of tasks and threads of a foreign partition in the system; a revelation that the operating system as separation kernel should prevent.

6.5.6 Test Drivers

In this section we address the problem to compile "abstract test-drivers" as described in the previous sections into concrete code and code instrumentations that actually execute these tests.

HOL-TestGen can generate test scripts (recall Figure 1.1) in SML, Haskell, Scala and F#. For our application, we generate SML test scripts and use MLton (www.mlton.org) for building the test executable: MLton 1. provides a foreign function interface to C and 2. is easily portable to small POSIX system (it mainly requires a C compiler, libc, and libm).[1]

In more detail, we generate two SML structures *automatically* from the Isabelle theories. The first structure, called `Datatypes`, contains the datatypes that are used by the interface of the SUT. In our example, this includes, e. g., `IPC_protocol` and `P4_IPC_call`. The second structure, called `TestScript`, contains a list of all generated test cases as well the *test oracle*, i. e., the algorithms necessary to decide if a test result complies to the specification or not. In addition, HOL-TestGen provides a test harness (as SML structure `TestHarness`) that 1. takes the list of test cases (from `TestScript`) and executes them on the SUT, 2. uses the test oracle (also from `TestScript`) to decide if the actual test results complies to the specification, and 3. provides statistics about the number of successful and failed tests as well as errors (e. g., unexpected exceptions) during test execution.

In addition, for testing C code, we need to provide a small SML structure (ca. 20 lines of code), called `Adapter`, that serves two purposes: 1. the configuration of the foreign function, e. g., the mapping from SML datatypes to C datatypes and 2. the concretization of abstractions to bridge the gap between an abstract test model and the concrete SUT.

An example for a concretization would be a test specification using an an enumeration to encode error states while the implementation uses an efficient encoding as bit vector. The `Adapter` structure only needs to be updated after significant changes to either the system specification or the system under test.

For testing concurrent, i. e., multi-threaded, programs we need to solve a particular challenge: *enforcing certain thread execution orders* (a certain scheduling) during test execution. There are, in principle, three different options available to control the scheduler during test execution: 1. instrumenting the SUT to make the thread switching deterministic and controllable, 2. using a deterministic scheduler that can be controlled by test driver, or 3. using the features of debuggers, such as the GNU debugger (gdb), for multi-threaded programs.

In our prototype for POSIX compliant systems, we have chosen the third option: we execute the SUT within a gdb session and we use the gdb to

[1] In our code generation setup, we avoid the use of the SML datatype `Int.Inf` and, by this, we can remove the dependency on the GNU multi-precision library (libgmp).

switch between the different threads in a controlled way. We rely on two features of gdb (thus, out approach can be applied to any other debugger with similar features), namely: 1. the possibility to attach to break points in the object code scripting code that is executed if a break point is reached and 2. the complete control of the threading, i. e., gdb allows to switch explicitly between threads while ensuring that only the currently active thread is executed (using the option `set scheduler-locking on`).

This approach has the advantage that we neither need to modify the SUT nor do we need to develop a custom scheduler. We only need to generate a configuration for controlling the debugger. The necessary gdb command file is generated automatically by HOL-Testgen based on a mapping of the abstract thread switching points to break points in the object code. The break points at the entry points allows us to control the thread creation, while the remaining break points allow us to control the switching between threads. Thus, we only need the SUT compiled in debugging mode and this mapping. In this sense, we still have a "black-box" testing approach.

Moreover, Using gdb together with `taskset`, we ensure that all threads are executed on the same core; in our application, we can accept that the actual execution in gdb changes the timing behavior. Moreover, we assume a sequential memory model, so our approach does not cover TLB-related race conditions occurring in multi-core CPU's.

A note on testing small embedded systems and low-level operating system code. This setup works well for mid-size embedded systems to large systems using standard desktop or server operating systems. It does not work for small embedded systems or for testing small operating system kernels or hypervisors. Such system often to neither provide a rich enough `libc` (or `libm`) nor enough system resources that allows to run the complete test driver on the system under test. For such systems, we envision a host-target setup, where only a very small target library needs to be ported to the target system. This target library serves mainly two purposes: 1. stimulate, remotely controlled from the host system, the functions under test and 2. collect the test result and report it back to the host system. All expensive computation such as comparing test results, creating statistics are executed on the host system.

Finally, for small systems it might be necessary to develop a custom scheduler, e. g., similar to [MQB07], to control the execution order of multi-threaded programs.

6.5.7 Experimental Results

In this section we will discuss our test experiences, the obtained results and the different problems encountered. The table Table 6.1 represent 5^2 different test specifications related to PikeOS IPC , i. e. test scenarios for PikeOS IPC API, and also the statistics related to the application of the different steps of our test generation process on these scenarios. Four columns are distinguished in Table 6.1:

1. **SE**: is the step related to the symbolic execution process. During this step the derived symbolic execution rules related to PikeOS IPC are applied on the scenario.

2. **Norm**: represent the step of our normalization process. During this step we apply tactics like `simp` and other derived rules from the model in order to eliminate contradictory proof goals resulting from the SE step.

3. **TT**: is the step of the generation of the test theorem. During this step we use an HOL-TestGen tactic to determine the PO and to introduce uniformity testing hypotheses (recall subsection 2.2.2) on the different proof goals resulting from the Norm step. This step can be seen as a preparatory step for the data selection process.

4. **TD**: represent the step of test data selection. During this step we send the POs in the test theorem to constraint-solvers. Also, after that a given solver choose a model for the POs an Isabelle proof is mandatory in order to make sure that the chosen model satisfies the PO. We have to notice that, for simple models, the process of proving the satisfaction of the PO by the chosen model, is done automatically by an Isabelle tactic but, for complicated models such as PikeOS model, where its symbolic execution results with complicated predicated defined around abstract types, e. g. predicate around our memory model, the proofs need to be done manually. This does not mean that the process can not be automated, but at the moment, we do not have the set of lemmas and the corresponding tactics that help to get a such automatic setup.

Each column in Table 6.1 is composed of two other columns. The columns named *Num* contain the number of outputs from each step of the generation process, and the columns *Time* contain the duration of the step by minutes. The scenarios Sc1 and Sc2 contain the value *undet* in their columns, it means that we did not manage to finish the steps of the generation and the experience is done for these scenarios. The judgement *undet* is different from the

[2] actually we designed 38 scenario, we did not finish all the experiments at submission time, further explanation are presented in the sequel.

judgement represented by the symbol −, also contained in the table. The judgement *undet* is applied to an experience where our process of test generation had failed in a given step, and we are not trying to fix the failed part because, the fixes depends on major changes in the various levels of the tool-chain. The judgement − is applied on an experience which is not finished yet, i. e. we do not have the results of all the steps of the process but, finishing the experience depends on manageable technical problems [3].

Note that the execution of the steps related to the test generation process is sequential. Thus, if the current step fails the next one can not be executed. For example during the scenario Sc1, we had derived actually 69984 symbolic test cases in 2 hours for 1 input sequence that represent an IPC communication (recall subsection 6.5.1) but, we did not manage to normalize a such proof context with a such size, which means that all remaining steps of the process can not be performed because they all depend of the outputs from the Norm step.

As explained in subsection 6.5.4, the generation of 69984 symbolic test cases does not necessarily mean that all the cases, represented by proof goals, are feasible. We have to normalize the proof goals in order to eliminate the contradictory ones. Even if we have managed to normalize a proof context with a such size, we still need to find models for the different normalized goals and prove that, the chosen models satisfy the POs. While the fact of generating almost 70000 goals using our symbolic approach in only 2 hours can be seen as an impressive result, we have failed during the normalization process, and this come back to:

1. **The model.** the model of PikeOS IPC is heavy, and this because of the branching in the atomic actions, especially the PREP action.

2. **The way of modeling.** it is the main influential factor. We believe that some changes on the way of modeling can help to make the normalization process lighter. e. g. the definition of meta-predicates that characterize feasible paths only, or at least the elimination of the most of infeasible paths, and accordingly, the definition of the corresponding symbolic execution rules, can actually result with an optimized proof context after the SE step.

In order to execute our tool-chain from top to bottom we have tried other test scenarios to avoid the previous cited problems. For example, the scenario Sc2 is similar to the scenario Sc1 but, without including the PREP stage in the input sequence that represent 1 IPC communication. From Sc2, we had derived 1973 symbolic test cases in 2 minutes (which is another impressive result). After 6 hours of normalization process, 27 abstract test case remained. But still we did not manage to get automatically models for the 27

[3] At submission time of this document, we had managed to finish only 4 experiences.

abstract test cases, and this because of a failure from the constraint-solvers, such `smt`, to provide a solution for complicated POs. The failure come back mainly to missing lemmas used as a configuration (recall last paragraphs in subsection 6.5.4) for the constraint solvers and not to the constraints-solver design.

For the scenarios Sc3 to Sc5, we have tried another approach in order to deal with the previous cited problems and also to generate test cases that cover communications with PREP action. Basically the approach is based on a technique that, allows to force a given execution path from the possible ones, resulting from the execution of the PREP action. Actually, after the execution of a PREP action, 6 execution paths are possible (see the symbolic execution rule for PREP action in section O). Since we have 2 PREP actions in the head of a sequence that represent 1 IPC communication, all possible execution paths related to the 2 PREP actions is equal to 6×6. Actually, the 2 PREP actions are derived from: the ipc send system call for the PREP SEND action, and the ipc receive system call for PREP RECV. Each system call is executed by a thread. Instead to opt for a standard execution of the 2 PREP actions with rules that simulate all possible executions paths like we did in Sc1, we had opted for rules that force one execution path inside a test scenario. In order to cover all paths, we had designed 36 scenarios, each scenario force a given execution path during the PREP stage. Because we do not have any problems for executing the other actions which are different from PREP, we used the standard rules for their symbolic execution.

In order to apply this new tchnique to our scenarios, new symbolic execution rules were designed to cope with the explosion in the number of the abstract test cases, which influence negatively our normalization process. For example, in the scenario Sc3 we had derived 2 new symbolic execution rules for PREP actions. Each rule characterize one execution path by assuming that the path-predicate that describe the execution path is true. The symbolic execution rules used to simulate the the behavior of the actions PREP_SEND and PREP_RECV in the scenario Sc3 are:

```
lemma abort_prep_send_HOL_elim21'_factor:
  assumes valid_exec:
   (σ ⊨(outs ←(mbind ((IPC PREP (SEND caller partner msg))#S)
         (abort_lift exec_action_id_Mon)); P outs))
  and in_err_exec1: caller ∈dom (act_info (th_flag σ))
  and in_err_exec:
      (σ ⊨(outs ←(mbind S(abort_lift exec_action_id_Mon));
             P (get_caller_error caller σ# outs))) ⟹Q

  shows Q
  apply (insert valid_exec)
  apply (elim abort_prep_send_mbindFSave_E')
  apply (simp add: in_err_exec)
  apply (simp add: in_err_exec1)+
  done
```

```
lemma abort_prep_recv_HOL_elim21'_factor:
  assumes valid_exec:
  (σ ⊨(outs ←(mbind ((IPC PREP (RECV caller partner msg))#S)
        (abort_lift exec_action_id_Mon)); P outs))
  and in_err_exec1: caller ∈dom (act_info (th_flag σ))
  and in_err_exec:
    (σ ⊨(outs ←(mbind S(abort_lift exec_action_id_Mon));
                              P (get_caller_error caller σ
# outs))) ⟹Q

   shows Q
  apply (insert valid_exec)
  apply (elim abort_prep_recv_mbindFSave_E')
  apply (simp add: in_err_exec)
  apply (simp add: in_err_exec1)+
  done
```

Of course the path-predicate `in_err_exec1` must be expressed also in the test specification Sc3. This predicate express the fact that the caller of the action (the caller of PREP SEND and also the caller of PREP RECV), was in an error-state (recall subsection 6.3.4).

From another side, we did not manage to execute the generated tests on PikeOS sources, for confidentiality reasons. In order to evaluate our approach we had implemented a PikeOS IPC-like environment using POSIX implementation. We had managed to execute 2 scenarios on this PikeOS demonstrator. Of course, when the state of the PikeOS demonstrator is initialised correctly our tests did not found any bugs. If the state is not ini-

Scenarios	SE		Norm		TT		TD	
	Num	Time	Num	Time	Num	Time	Num	Time
Sc1	69984	120	*undet*	*undet*	*undet*	*undet*	*undet*	*undet*
Sc2	1973	2	27	360	1	162	*undet*	*undet*
Sc3	1973	2	2	0.01	1	120	2080	0.23
Sc4	1973	2	-	-	-	-	-	-
Sc5	1973	2	-	-	-	-	-	-

Table 6.1: Statistics for our TestGen Process

tialised correctly our generated tests detect the bugs. Finally, we still have problems to define a program that initialise automatically the state of the demonstrate and bring it to the same value generated by the model. At the moment this step is done manually, and this due to some technical challenges like, how to export or import the values of a static array defined on C-level to the sml-level. Finally, another technical challenge is that GDB can not run an executable containing a `Main.sml` function defined in `sml` language. In order to deal with this problem, we have to define a `Main.c` function on C-level and call our `harness.sml` inside the `Main.c`, and this using the foreign function interface of `MLton`.

6.6 Conclusion

6.6.1 Related Work.

There is a wealth of approaches for tests of behavioral models; they differ in the underlying modeling technique, the testability and test hypothesis', the test conformance relation etc.; in subsection 2.2.3 we mention a few. Unfortunately, many works make the underlying testability hypothesis' not explicit which makes a direct comparison difficult and somewhat vague. For the space of testability assumptions used here (the system is input-output deterministic, is adequately modeled as under-specified deterministic system, synchronous coupling between tester and SUT suffices), to the best of our knowledge, our approach is unique in its integrated process from theory, modeling, symbolic execution down to test-driver generation.

With respect to the test-driver approach, this work undeniably owes a lot Microsoft's CHESS project [MQB07], which promoted the idea to actually control the scheduler of real systems and use partial-order reduction techniques to test systematically concurrent executions for races in applications of realistic size (e. g., IE, Firefox, Apache). For our approach, controlling the scheduler is the key to justify the presentation of the system as under-specified deterministic transition function.

6.6.2 Conclusion and Future Work.

We see several conceptual and practical advantages of a *monadic approach* to sequence testing:

1. a monadic approach resists the tendency to surrender to finitism and constructivism at the first-best opportunity; a tendency that is understandably wide-spread in model-checking communities,

2. it provides a sensible shift from syntax to semantics: instead of a first-order, intentional view in *nodes* and *events* in automata, the heart of the calculus is on *computations* and their *compositions*,

3. the monadic theory models explicitly the difference between input and output, between data under control of the tester and results under control of the SUT,

4. the theory lends itself for a theoretical and practical framework of numerous conformance notions, even non-standard ones, and which gives

5. ways to new calculi of symbolic evaluation enabling symbolic states (via invariants) and input events (via constraints) as well as a seamless, theoretically founded transition from system models to test-drivers.

We see several directions for future work: On the model level, the formal theory of sequence testing (as given in the HOL-TestGen library theories `Monad.thy` and `TestRefinements.thy`) providing connections between monads, rules for test-driver optimization, different test refinements, etc., is worth further development. On a test-theoretical level, our approach provides the basis for a comparison on test-methods, in particular ones based on different testability hypothesis'.

Pragmatically, our test driver setup needs to be modified to be executable on the PikeOS system level. For this end, we will need to develop a host-target setup (see subsection 6.5.6). Finally, we are interested in extending our techniques to actually test information flow properties; since error-codes in applications may reveal internal information of partitions (as, for example, the number of its tasks and threads), this seems to be a rewarding target. For this purpose, not only action sequences need to be generated during the constraint solving process, but also (abstract) VMITs.

Part III

Conclusions

Conclusions and Future Works

7.1 Summary

In the different chapters of this, we introduced thesis a test and proof environment for the specification, deductive verification and testing of concurrent programs. Our approach relies on the theorem proving Isabelle/**hol!**. In the context of this Ph.D thesis, the architecture, features, tools and the underlying methodology of Isabelle were also presented. We believe to have justified our claim that was, if correctly used, Isabelle can be trusted to a significantly higher extent than conventional software used in certification processes and test generation based on symbolic execution. The main problematic tackled by our work was the generation of tests for the certification of complex concurrent systems such operating systems. Our solution was, first of all, the proposition of a monad based test theory for the specification of a such system. Afterwards, we had used symbolic execution approach, based on theorem proving environment in order to generate test cases. Our contributions, results and achievement are:

Isabelle/hol! in Certification Processes

In section 2.3, we have presented the Isabelle/**hol!** system and pointed out the essential arguments, why by a particular combination of system-architecture and methodology, the system is suited to give the currently highest possible

guarantee on a formal proof in particular and a logical theory development in general. In a sense, Isabelle/**hol!** offers the same guarantees for logical systems as Coq, and in some sense better guarantees than, for example, the B method or model-checkers like FDR. Isabelle/hol! is therefore a natural choice for evaluations in the higher certification levels EAL5 to EAL7 in the Common Criteria (CC). If the methodological side-conditions are respected which can be reduced essentially to a number of syntactic checks, the formal consistency of the entire certification document containing formal specifications, proofs of consistency and the proofs of security properties, refinement-proofs between the different abstraction layers, and finally test-case generations as well as test-results can be guaranteed, and the evaluator can therefore concentrate on the more fundamental questions: does the model represent the right thing? are the modeling assumptions justified?

A Monad Based Testing Theory

Our framework is equipped with a specification language (see chapter 4) based on monads that contains important definitions for testing and symbolic execution activities. The expressive power of our specification language was was highlighted by an isomorphism between the automata world and monads world. A set of generic symbolic execution rules, for the defined monad operators was also introduced. Unlike to IO-Automata based specification, it turns out that symbolic execution based on monads specifications and its representation in the **hol!** language can cope with the large state space; and that was confirmed by the results that we have got from our case studies, where our approach was applied on on two complex systems.

Sequence Testing For Concurrent Complex Systems

In order to optimize the symbolic execution process for our test specifications, especially for the case of sequence test scenarios of concurrent systems, an approach based on the notion of coverage criteria was proposed in section 4.4. On the technical side, an approach to build automated test drivers for testing non-deterministic system executions was proposed in section 4.7.

Testing VAMP Microprocessor

As a case study, and in order to confirm the efficiency of our test framework, we presented in chapter 5, an approach for testing the conformance of a processor with respect to an abstract model that captures the instruction set (i. e., the assembly-level) of the processor. This abstraction level is of a particular importance for, first, it is the level of detail that is usually available for commercial off-the-shelf (COTS) processors and, second, it is the target level of high-level compilers.

Testing PikeOS System

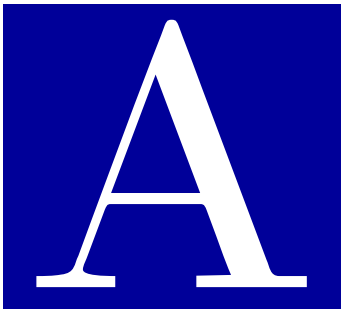
Another achievement of our work was presented in chapter 6. It consist of a case study, containing the formalization and test case generation for the complex operating system PikeOS. The approach allows for testing relatively fine-grained concurrency of atomic actions, which are actually related to system calls of an L4-like micro-kernel. During this case study we focused on the IPC API. The case study was another confirmation of the expressive power and the efficiency of theorem proving based test framework. Especially if the framework is combined with a monad based testing theory.

7.2 Futur Works

We see several directions for future works: On the model level, the formal theory of sequence testing (as given in the HOL-TestGen library theories Monad.thy and TestRefinements.thy) providing connections between monads, rules for test-driver optimization, different test refinements, etc., is worth further development. On a test-theoretical level, our approach provides the basis for a comparison on test-methods, in particular ones based on different testing hypotheses and a bit indirectly even the underlying hypotheses of testability. Pragmatically, our test driver setup needs to be modified to be executable on systems such as PikeOS. For this end, we will need to develop a host-target setup approach, that can cope with restrictions of low level system code, e. g. limited libraries, limited access to IO. Finally, we are interested in extending our techniques to actually test security properties such as information flow properties; since error-codes in applications may reveal internal information of partitions (as, for example, the number of its tasks and threads), this seems to be a rewarding target. For this purpose, not only action sequences need to be generated during the constraint solving process, but also (abstract) VMITs.

Part IV

PikeOS IPC Model



Isabelle sources

theory *TypeSchemes*
imports *Main*
begin

A HOL representation of PikeOS Datatypes

A.1 kernel state

record (*'resource*, *'thread-id*,*'thread*, *'sp-th-th*, *'sp-th-res*,*'errors*) *kstate* =
resource :: *'resource* — system ressources: memory, files..
current-thread :: *'thread-id* — a thread in the execution context..
thread-list :: *'thread* — list of threads in the system.
communication-rights :: *'sp-th-th* — security policy between threads..
access-rights :: *'sp-th-res* — security policy between threads and ressources..

error-codes :: *'errors* — error returned if a system call is aborted..

A.2 atomic actions

Atomic actions can be seen as instructions which can not be interrupted by the system scheduler during there execution. Each API has its own set of atomic actions.

datatype (*'ipc-stage*, *'ipc-direction*) $action_{ipc}$ =

IPC 'ipc-stage 'ipc-direction

datatype (*'mem-param1*, *'mem-param2*) $action_{mem}$ =
MEM 'mem-param1 'mem-param2

datatype (*'evn-param1*, *'evn-param2*) $action_{evn}$ =
EVN 'evn-param1 'evn-param2

datatype (*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *action* =
$atom_{ipc}$ (*'ipc-stage*, *'ipc-direction*) $action_{ipc}$
| $atom_{mem}$ (*'mem-param1*, *'mem-param2*) $action_{mem}$
| $atom_{evn}$ (*'evn-param1*, *'evn-param2*) $action_{evn}$

A.3 traces

A trace is sequence of atomic actions..

— An IPC actions trace

type-synonym (*'ipc-stage*, *'ipc-direction*) $trace_{ipc}$ =
(*'ipc-stage*, *'ipc-direction*) $action_{ipc}$ *list*

— A memory actions IPC trace

type-synonym (*'mem-param1*, *'mem-param2*) $trace_{mem}$ =
(*'mem-param1*, *'mem-param2*) $action_{mem}$ *list*

— An event actions trace

type-synonym (*'evn-param1*, *'evn-param2*) $trace_{evn}$ =
(*'evn-param1*, *'evn-param2*) $action_{evn}$ *list*

— A trace that contain all atomic actions

type-synonym (*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *trace* =
(*'ipc-stage*, *'ipc-direction*,*'mem-param1*, *'mem-param2*, *'evn-param1*, *'evn-param2*) *action list*

A.4 Threads

A thread is the smallest entity in the operating system.

record (*'th-id*,*'thstate*,*'stipc*,*'vadress*,*'cpartner*) *thread* =
thread-id :: *'th-id*
th-state :: *'thstate*
th-ipc-st :: *'stipc*
own-vmem-adr :: *'vadress*
cpartner :: *'cpartner*

end

theory *SharedMemoryNew*
imports *Main*
begin

B Shared Memory Model

B.1 Prerequisites

Prerequisite: a generalization of *fun-upd-def*: $?f(?a := ?b) \equiv \lambda x.\ if\ x = ?a\ then\ ?b\ else\ ?f\ x$. It represents updating modulo a sharing equivalence, i.e. an equivalence relation on parts of the domain of a memory.

definition *fun-upd-equivp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$ **where**
fun-upd-equivp eq f a b = $(\lambda x.\ if\ eq\ x\ a\ then\ b\ else\ f\ x)$

— This lemma is the same as *Fun.fun-upd-same*: $(?f(?x := ?y))\ ?x = ?y$; applied on our genralization *fun-upd-equivp ?eq ?f ?a ?b* = $(\lambda x.\ if\ ?eq\ x\ ?a\ then\ ?b\ else\ ?f\ x)$ of $?f(?a := ?b) \equiv \lambda x.\ if\ x = ?a\ then\ ?b\ else\ ?f\ x$. This proof tell if our function *fun-upd-equivp op* = *f x y* is equal to *f* this is equivalent to the fact that *f x* = *y*

lemma *fun-upd-equivp-iff*: $((fun\text{-}upd\text{-}equivp\ (op =)\ f\ x\ y) = f) = (f\ x = y)$
by (*simp add* :*fun-upd-equivp-def*, *safe*, *erule subst*, *auto*)

— Now we try to proof the same lemma applied on any equivalent relation *equivp eqv* instead of the equivalent relation *op* =. For this case, we had split the lemma to 2 parts. the lemma *fun-upd-equivp-iff-part1* to proof the case when *eq* (*f a*) *b* $\longrightarrow$ *eq* (*fun-upd-equivp eqv f a b z*) (*f z*), and the second part is the lemma *fun-upd-equivp-iff-part2* to proof the case *equivp eqv* $\Longrightarrow$ *fun-upd-equivp eqv f a b* = *f* $\longrightarrow$ *f a* = *b*.

lemma *fun-upd-equivp-iff-part1*:
equivp R $\Longrightarrow$ $(\bigwedge z.\ R\ x\ z \Longrightarrow R\ (f\ z)\ y) \Longrightarrow R\ (fun\text{-}upd\text{-}equivp\ R\ f\ x\ y\ z)\ (f\ z)$
by (*auto simp*: *fun-upd-equivp-def Equiv-Relations.equivp-reflp Equiv-Relations.equivp-symp*)

lemma *fun-upd-equivp-iff-part2*: *equivp R* $\Longrightarrow$ *fun-upd-equivp R f x y* = *f* $\longrightarrow$ *f x* = *y*
apply (*simp add* :*fun-upd-equivp-def*, *safe*)
apply (*erule subst*, *auto simp*: *Equiv-Relations.equivp-reflp*)
done

— Just anotther way to formalise *equivp ?R* $\Longrightarrow$ *fun-upd-equivp ?R ?f ?x ?y* = *?f* $\longrightarrow$ *?f ?x* = *?y* without using the strong equality

lemma *equivp R* $\Longrightarrow$ ($\bigwedge z$. *R x z* $\Longrightarrow$ *R* (*fun-upd-equivp R f x y z*) (*f z*)) $\Longrightarrow$ *R y* (*f x*)
by (*simp add*: *fun-upd-equivp-def Equiv-Relations.equivp-symp equivp-reflp*)

— this lemma is the same in $[\![$*equivp ?R*; $\bigwedge z$. *?R ?x z* $\Longrightarrow$ *?R* (*?f z*) *?y*$]\!] \Longrightarrow$ *?R* (*fun-upd-equivp ?R ?f ?x ?y ?z*) (*?f ?z*) where *op* = is generalized by another equivalence relation

lemma *fun-upd-equivp-idem*: *f x* = *y* $\Longrightarrow$ (*fun-upd-equivp* (*op* =) *f x y*) = *f*
by (*simp only*: *fun-upd-equivp-iff*)

lemma *fun-upd-equivp-triv* : *fun-upd-equivp* (*op* =) *f x* (*f x*) = *f*
by (*simp only*: *fun-upd-equivp-iff*)

— This is the generalization of *fun-upd-equivp op* = *?f ?x* (*?f ?x*) = *?f* on a given equivalence relation

lemma *fun-upd-equivp-triv-part1* :
equivp R $\Longrightarrow$ ($\bigwedge z$. *R x z* $\Longrightarrow$*fun-upd-equivp* (*R'*) *f x* (*f x*) *z*) $\Longrightarrow$ *f x*
apply (*auto simp*:*fun-upd-equivp-def*)
apply (*metis equivp-reflp*)
done

lemma *fun-upd-equivp-triv-part2* :
equivp R $\Longrightarrow$ ($\bigwedge z$. *R x z* $\Longrightarrow$ *f z*) $\Longrightarrow$ *fun-upd-equivp* (*R'*) *f x* (*f x*) *x*
by (*simp add*:*fun-upd-equivp-def equivp-reflp split*: *split-if*)

lemma *fun-upd-equivp-apply* [*simp*]:
(*fun-upd-equivp* (*op* =) *f x y*) *z* = (*if z* = *x then y else f z*)
by (*simp only*: *fun-upd-equivp-def*)

— This is the generalization of *fun-upd-equivp op* = *?f ?x ?y ?z* = (*if ?z* = *?x then ?y else ?f ?z*) with e given equivalence relation and not only with *op* =

lemma *fun-upd-equivp-apply1* [*simp*]:
equivp R $\Longrightarrow$(*fun-upd-equivp R f x y*) *z* = (*if R z x then y else f z*)
by (*simp add*: *fun-upd-equivp-def*)

lemma *fun-upd-equivp-same*: (*fun-upd-equivp* (*op* =) *f x y*) *x* = *y*
by (*simp only*: *fun-upd-equivp-def*)*simp*

— This is the generalization of *fun-upd-equivp op* = *?f ?x ?y ?x* = *?y* with a given equivalence relation

lemma *fun-upd-equivp-same1*: *equivp R* $\Longrightarrow$ (*fun-upd-equivp R f x y*) *x* = *y*
by (*simp add*: *fun-upd-equivp-def equivp-reflp*)

For the special case that @term eq is just the equality @term "op =", sharing update and classical update are identical.

lemma *fun-upd-equivp-vs-fun-upd*: (*fun-upd-equivp* (*op* =)) = *fun-upd*
by(*rule ext*, *rule ext*, *rule ext*,*simp add*:*fun-upd-def fun-upd-equivp-def*)

B.2 Definition of the shared-memory type

typedef ('*α*, '*β*) *memory* = {(*σ*::'*α* ⇀'*β*, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)}
proof
show (*Map.empty*, (*op* =)) ∈ *?memory*
by (*auto simp*: *identity-equivp*)
qed

fun *memory-inv* :: ('*a* ⇒ '*b option*) × ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool*
where *memory-inv* (*Pair f R*) = (*equivp R* ∧ (∀ *x y*. *R x y* ⟶ *f x* = *f y*))

lemma *Abs-Rep-memory* [*simp*]:*Abs-memory* (*Rep-memory σ*) = *σ*
by (*simp add*:*Rep-memory-inverse*)

lemma *memory-invariant* [*simp*]:
memory-inv σ-rep = (*Rep-memory* (*Abs-memory σ-rep*) = *σ-rep*)
using *Rep-memory* [*of Abs-memory σ-rep*] *Abs-memory-inverse mem-Collect-eq*
prod-caseE prod-caseI2 memory-inv.simps
by *smt*

lemma *Pair-code-eq* :
Rep-memory σ = *Pair* (*fst* (*Rep-memory σ*)) (*snd* (*Rep-memory σ*))
by (*simp add*: *Product-Type.surjective-pairing*)

lemma *snd-memory-equivp* [*simp*]: *equivp*(*snd*(*Rep-memory σ*))
by(*insert Rep-memory*[*of σ*], *auto*)

B.3 Operations on Shared-Memory

setup-lifting *type-definition-memory*
abbreviation *mem-init* :: ('*a* ⇒ '*b option*) × ('*a* ⇒ '*a* ⇒ *bool*)
where
mem-init ≡ (*Map.empty*, (*op* =))

lemma *memory-init-eq-sound*:
mem-init ∈ {(*σ*, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)}
proof −
obtain *mem* **and** *R*
where *Pair*: (*mem*, *R*) =*mem-init* **and** *Eq*: *equivp R*
using *identity-equivp* **by** *auto*
have *D1*: *R* = (*op* =)
and *D2*: *mem* = *Map.empty*
using *Pair prod.inject*
by *auto*
moreover have *inv-part2*: ∀ *x y* . *R x y* ⟶ *mem x* = *mem y*

unfolding *D1 D2* **by** *auto*
ultimately show *?thesis*
using *Eq Abs-memory-cases Pair-inject Rep-memory-cases Rep-memory-inverse*

identity-equivp memory-inv.elims(3) memory-invariant
by *auto*
qed

lift-definition *init* :: ('*α*, '*β*) *memory*
is *mem-init* :: ('*α* ⇒ '*β option*) × ('*α* ⇒ '*α* ⇒ *bool*)
using *memory-init-eq-sound* **by** *simp*

value *init*::(*nat*,*int*)*memory*
value *map* (*λx*. *the* (*fst* (*Rep-memory init*)*x*)) [*1* .. *10*]
value *take* (*10*) (*map* (*Pair Map.empty*) [(*op* =)])
value *replicate 10 init*
term *Rep-memory σ*
term [(*σ*::*nat* ⇀ *int*, *R*)<−*xs* . *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)]

definition *init-mem-list* :: '*α list* ⇒ (*nat*, '*α*) *memory*
where *init-mem-list s* = *Abs-memory* (*let h* = *zip* (*map nat* [*0* .. *int*(*length s*)]) *s*
in foldl (*λx* (*y*,*z*). *fun-upd x y* (*Some z*))
Map.empty h,
op =)

Memory Read Operation

definition *lookup* :: ('*α*, '*β*) *memory* ⇒ '*α* ⇒ '*β* (**infixl** $ *100*)
where *σ* $ *x* = *the* (*fst* (*Rep-memory σ*) *x*)

Memory Update Operation

fun *Pair-upd-lifter*:: ('*a* ⇒ '*b option*) × ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*b* ⇒
('*a* ⇒ '*b option*) × ('*a* ⇒ '*a* ⇒ *bool*)
where *Pair-upd-lifter* ((*f*, *R*)) *x y* = (*fun-upd-equivp R f x* (*Some y*), *R*)

lemma *update_sound'*:
assumes *σ* ∈ {(*σ*, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)}
shows *Pair-upd-lifter σ x y* ∈ {(*σ*, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)}
proof −
obtain *mem* **and** *R*
where *Pair*: (*mem*, *R*) = *σ* **and** *Eq*: *equivp R* **and** *Mem*: ∀ *x y* . *R x y* ⟶ *mem x* = *mem y*
using *assms equivpE* **by** *auto*
obtain *mem′* **and** *R′*
where *Pair′*: (*mem′*, *R′*) = *Pair-upd-lifter σ x y*

using *surjective-pairing* **by** *metis*
have *Def1*: *mem′* = *fun-upd-equivp R mem x* (*Some y*)
and *Def2*: *R′* = *R*
using *Pair Pair′* **by** *auto*
have *Eq′*: *equivp R′*
using *Def2 Eq* **by** *auto*
moreover have ∀ *y z* . *R′ y z* ⟶ *mem′ y* = *mem′ z*
using *Mem equivp-symp equivp-transp*
unfolding *Def1 Def2* **by** (*metis Eq fun-upd-equivp-def*)
ultimately show *?thesis*
using *Pair′* **by** *auto*
qed

lift-definition *update* :: (′α, ′β) *memory* ⇒′α ⇒ ′β ⇒ (′α, ′β) *memory* (- ′(- :=$ -′) *100*)
is *Pair-upd-lifter*
using *update_sound′*
by *simp*

lemma *update′*: σ (*x* :=$ *y*) = *Abs-memory* (*fun-upd-equivp* (*snd* (*Rep-memory* σ))
(*fst* (*Rep-memory* σ)) *x* (*Some y*), (*snd* (*Rep-memory* σ)))
using *Rep-memory-inverse surjective-pairing Pair-upd-lifter.simps update.rep-eq*
by *metis*

fun *update-list-rep* :: (′α ⇁ ′β) × (′α ⇒ ′α ⇒ *bool*) ⇒ (′α × ′β)*list* ⇒
(′α ⇁ ′β) × (′α ⇒ ′α ⇒ *bool*)
where *update-list-rep* (*f*, *R*) *nlist* = (*foldl* (λ(*f*, *R*)(*addr*,*val*). *Pair-upd-lifter* (*f*, *R*) *addr val*)
(*f*, *R*)
nlist)

lemma *update-list-rep-p*:
assumes *1*: *P* σ
and *2*: ⋀*src dst* σ. *P* σ ⟹ *P* (*Pair-upd-lifter* σ *src dst*)
shows *P* (*update-list-rep* σ *list*)
using *1 2*
apply (*induct list arbitrary*: σ)
apply (*force*,*safe*)
apply (*simp del*: *Pair-upd-lifter.simps*)
using *surjective-pairing*
apply *metis*
done

lemma *update-list-rep-sound*:
assumes *1*: σ ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}

shows *update-list-rep* σ (*nlist*) $\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
using *1*
apply (*elim update-list-rep-p*)
apply (*erule update_sound'*)
done

lift-definition *update-list* :: ('α, 'β) *memory* $\Rightarrow$ ('$\alpha \times$ 'β)*list* $\Rightarrow$ ('α, 'β) *memory* (**infixl** '/:=$_\$$ *30*)
is *update-list-rep*
using *update-list-rep-sound* **by** *simp*

lemma *update-list-Nil*[*simp*]: (σ /:=$_\$$ []) = σ
unfolding *update-list-def*
by(*simp*,*subst surjective-pairing*[*of Rep-memory* σ],*subst update-list-rep.simps*, *simp*)

lemma *update-list-Cons*[*simp*] : (σ /:=$_\$$ ((*a*,*b*)#*S*)) = (σ(*a* :=$_\$$ *b*) /:=$_\$$ *S*)
unfolding *update-list-def*
apply(*simp*,*subst surjective-pairing*[*of Rep-memory* σ],*subst update-list-rep.simps*, *simp*)
apply(*subst surjective-pairing*[*of Rep-memory* (σ (*a* :=$_\$$ *b*))],*subst update-list-rep.simps*, *simp*)
apply(*simp add*: *update-def*)
apply(*subst Abs-memory-inverse*)
apply (*metis* (*lifting*, *mono-tags*) *Rep-memory update_sound'*)
by *simp*

Type-invariant:

lemma *update_sound*:
assumes *Rep-memory* σ = (σ', *eq*)
shows (*fun-upd-equivp eq* σ' *x* (*Some y*), *eq*) $\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
using *assms insert Rep-memory*[*of* σ]
apply(*auto simp*: *fun-upd-equivp-def*)
apply(*rename-tac xa xb*, *erule contrapos-np*)
apply(*rule-tac R*=*eq* **and** *y*=*xa* **in** *equivp-transp*,*simp*)
apply(*erule equivp-symp*, *simp-all*)
apply(*rename-tac xa xb*, *erule contrapos-np*)
apply(*rule-tac R*=*eq* **and** *y*=*xb* **in** *equivp-transp*,*simp-all*)
done

Memory Transfer Based on Sharing Transformation

fun *transfer-rep* :: ('$\alpha \rightharpoonup$ 'β) $\times$ ('$\alpha \Rightarrow$ '$\alpha \Rightarrow$ *bool*) $\Rightarrow$ '$\alpha \Rightarrow$ '$\alpha \Rightarrow$ ('$\alpha \rightharpoonup$ 'β) $\times$ ('$\alpha \Rightarrow$ '$\alpha \Rightarrow$ *bool*)
where *transfer-rep* (*m*, *r*) *src dst* = (*m o* (*id* (*dst* := *src*)),
(λ *x y* . *r* ((*id* (*dst* := *src*)) *x*) ((*id* (*dst* := *src*)) *y*)))

lemma *transfer-rep-simp* :

transfer-rep X src dst = ((*fst X*) *o* (*id* (*dst* := *src*)),
(λ *x y* . (*snd X*) ((*id* (*dst* := *src*)) *x*) ((*id* (*dst* := *src*)) *y*)))
by(*subst surjective-pairing*[*of X*],*subst transfer-rep.simps*, *simp*)

lemma *transfer-rep-sound*:
assumes σ ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
shows *transfer-rep* σ *src dst* ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
proof −
obtain *mem* **and** *R*
where *P*: (*mem*, *R*) = σ **and** *E*: *equivp R* **and** *M*: ∀ *x y* . *R x y* ⟶ *mem x* = *mem y*
using *assms equivpE* **by** *auto*
obtain *mem′* **and** *R′*
where *P′*: (*mem′*, *R′*) = *transfer-rep* σ *src dst*
by (*metis surj-pair*)
have *D1*: *mem′* = (*mem o* (*id* (*dst* := *src*)))
and *D2*: *R′* = (λ *x y* . *R* ((*id* (*dst* := *src*)) *x*) ((*id* (*dst* := *src*)) *y*))
using *P P′* **by** *auto*
have *equivp R′*
using *E* **unfolding** *D2 equivp-def* **by** *metis*
moreover have ∀ *y z* . *R′ y z* ⟶ *mem′ y* = *mem′ z*
using *M* **unfolding** *D1 D2* **by** *auto*
ultimately show *?thesis*
using *P′* **by** *auto*
qed

lift-definition *transfer* :: ('α,'β)*memory* ⇒ 'α ⇒ 'α ⇒ ('α, 'β)*memory* (- '(- ⋈ -') [*0*,*111*,*111*]*110*)
is *transfer-rep*
using *transfer-rep-sound*
by *simp*

lemma *transfer-rep-sound2* :
transfer-rep (*Rep-memory* σ) *a b* ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
by (*metis* (*lifting*, *mono-tags*) *Rep-memory transfer-rep-sound*)

fun *share-list2* :: ('α, 'β) *memory* ⇒ ('α × 'α)*list* ⇒ ('α, 'β) *memory* (**infix** '/⋈ *60*)

where σ /⋈ S = (*foldl* (λ σ (a,b). (σ (a⋈b))) σ S)

lemma *sharelist2-Nil*[*simp*] : σ /⋈ [] = σ **by** *simp*

lemma *sharelist2-Cons*[*simp*] : σ /⋈ ((a,b)#S) = (σ(a⋈b) /⋈ S) **by** *simp*

fun *share-list-rep* :: ('α ⇁ 'β) × ('α ⇒ 'α ⇒ *bool*) ⇒ ('α × 'α)*list* ⇒
('α ⇁ 'β) × ('α ⇒ 'α ⇒ *bool*)
where *share-list-rep* (f, R) *nlist* =
(*foldl* (λ(f, R) (*src*,*dst*). *transfer-rep* (f, R) *src dst*) (f, R)
nlist)

fun *share-list-rep'* :: ('α ⇁ 'β) × ('α ⇒ 'α ⇒ *bool*) ⇒ ('α × 'α)*list* ⇒
('α ⇁ 'β) × ('α ⇒ 'α ⇒ *bool*)
where *share-list-rep'* (f, R) [] = (f, R)
| *share-list-rep'* (f, R) (n#*nlist*) = *share-list-rep'* (*transfer-rep*(f,R)(*fst* n)(*snd* n)) *nlist*

lemma *share-list-rep'-p*:
assumes *1*: P σ
and *2*: $\bigwedge$*src dst* σ. P σ ⟹ P (*transfer-rep* σ *src dst*)
shows P (*share-list-rep'* σ *list*)
using *1 2*
apply (*induct list arbitrary*: σ P)
apply *force*
apply *safe*
apply (*simp del*: *transfer-rep.simps*)
using *surjective-pairing*
apply *metis*
done

lemma *foldl-preserve-p*:
assumes *1*: P *mem*
and *2*: $\bigwedge$$y$ z *mem* . P *mem* ⟹ P (f *mem* y z)
shows P (*foldl* (λa (y, z). f *mem* y z) *mem list*)
using *1 2*
apply (*induct list arbitrary*: f *mem* , *auto*)
apply *metis*
done

lemma *share-list-rep-p*:
assumes *1*: P σ
and *2*: $\bigwedge$*src dst* σ. P σ ⟹ P (*transfer-rep* σ *src dst*)
shows P (*share-list-rep* σ *list*)
using *1 2*
apply (*induct list arbitrary*: σ)

apply *force*
apply *safe*
apply (*simp del*: *transfer-rep.simps*)
using *surjective-pairing*
apply *metis*
done

The modification of the underlying equivalence relation on adresses is only defined on very strong conditions — which are fulfilled for the empty memory, but difficult to establish on a non-empty-one. And of course, the given relation must be proven to be an equivalence relation. So, the case is geared towards shared-memory scenarios where the sharing is defined initially once and for all.

definition $update_R :: ('\alpha, '\beta)memory \Rightarrow ('\alpha \Rightarrow '\alpha \Rightarrow bool) \Rightarrow ('\alpha, '\beta)memory$ (- $:=_R$ - *100*)
where $\sigma :=_R R \equiv Abs\text{-}memory\ (fst(Rep\text{-}memory\ \sigma), R)$

definition $lookup_R :: ('\alpha, '\beta)memory \Rightarrow ('\alpha \Rightarrow '\alpha \Rightarrow bool)$ ($\$_R$ - *100*)
where $\$_R\ \sigma \equiv (snd(Rep\text{-}memory\ \sigma))$

lemma $update_R$-*comp*-$lookup_R$:
assumes *equiv* : *equivp R*
and *sharing-conform* : $\forall\ x\ y.\ R\ x\ y \longrightarrow fst(Rep\text{-}memory\ \sigma)\ x = fst(Rep\text{-}memory\ \sigma)\ y$
shows $(\$_R\ (\sigma :=_R R)) = R$
unfolding $lookup_R$-*def* $update_R$-*def*
by(*subst Abs-memory-inverse*, *simp-all add*: *equiv sharing-conform*)

B.4 Sharing Relation Definition

definition *sharing* :: $'\alpha \Rightarrow ('\alpha, '\beta)memory \Rightarrow '\alpha \Rightarrow bool$
((- *shares*()_/ -) [*201*, *0*, *201*] *200*)
where $(x\ shares_\sigma\ y) \equiv (snd(Rep\text{-}memory\ \sigma)\ x\ y)$

definition *Sharing* :: $'\alpha\ set \Rightarrow ('\alpha, '\beta)memory \Rightarrow '\alpha\ set \Rightarrow bool$
((- *Shares*()_/ -) [*201*, *0*, *201*] *200*)
where $(X\ Shares_\sigma\ Y) \equiv (\exists\ x{\in}X.\ \exists\ y{\in}Y.\ x\ shares_\sigma\ y)$

B.5 Properties on Sharing Relation

lemma *sharing-charn*:
equivp (*snd* (*Rep-memory* σ))
using *Rep-memory*[*of* σ]
unfolding *sharing-def*
by *auto*

lemma *sharing-charn'*:
assumes *1*: $(x\ shares_\sigma\ y)$

shows $(\exists R.\ equivp\ R \wedge R\ x\ y)$
by (*auto simp add*: *sharing-def snd-def equivp-def*)

lemma *sharing-charn2*:
shows$\exists x\ y.\ (\ equivp\ (snd\ (Rep\text{-}memory\ \sigma)) \wedge (snd\ (Rep\text{-}memory\ \sigma))\ x\ y)$
using *sharing-charn* [*THEN equivp-reflp*]
by (*simp*)*fast*

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is reflexive

lemma *sharing-refl*: $(x\ shares_{\sigma}\ x)$
using *insert Rep-memory*[*of* σ]
by (*auto simp*: *sharing-def elim*: *equivp-reflp*)

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is symetric

lemma *sharing-sym* [*sym*]:
assumes $x\ shares_{\sigma}\ y$
shows $y\ shares_{\sigma}\ x$
using *assms Rep-memory*[*of* σ]
by (*auto simp*: *sharing-def elim*: *equivp-symp*)

lemma *sharing-commute* : $x\ shares_{\sigma}\ y = (y\ shares_{\sigma}\ x)$
by(*auto intro*: *sharing-sym*)

— Lemma to show that $?x\ shares_{?\sigma}\ ?y \equiv snd\ (Rep\text{-}memory\ ?\sigma)\ ?x\ ?y$ is transitive

lemma *sharing-trans* [*trans*]:
assumes $x\ shares_{\sigma}\ y$
and $y\ shares_{\sigma}\ z$
shows $x\ shares_{\sigma}\ z$
using *assms insert Rep-memory*[*of* σ]
by(*auto simp*: *sharing-def elim*: *equivp-transp*)

lemma *shares-result*:
assumes $x\ shares_{\sigma}\ y$
shows $fst\ (Rep\text{-}memory\ \sigma)\ x = fst\ (Rep\text{-}memory\ \sigma)\ y$
using *assms*
unfolding *sharing-def*
using *Rep-memory*[*of* σ]
by *auto*

lemma *sharing-init*:

assumes *1*: $i \neq k$
shows $\neg(i\ shares_{init}\ k)$
unfolding *sharing-def init-def*
using *1*
by (*auto simp*: *Abs-memory-inverse identity-equivp*)

lemma *shares-init*[*simp*]: $(x\ shares_{init}\ y) = (x{=}y)$
unfolding *sharing-def init-def*
by (*metis init-def sharing-init sharing-def sharing-refl*)

lemma *sharing-init-mem-list*:
assumes *1*: $i \neq k$
shows $\neg(i\ shares_{init\text{-}mem\text{-}list\ S}\ k)$
unfolding *sharing-def init-mem-list-def*
using *1*
by (*auto simp*: *Abs-memory-inverse identity-equivp*)

definition *reset* :: $('\alpha, '\beta)\ memory \Rightarrow '\alpha\ set \Rightarrow ('\alpha, '\beta) memory$ (- '(*reset* -') *100*)
where $\sigma\ (reset\ X) = (let\ (\sigma', eq) = Rep\text{-}memory\ \sigma;$
$eq' = \lambda\ a\ b.\ eq\ a\ b \vee (\exists x{\in}X.\ eq\ a\ x \vee eq\ b\ x)$
in *if* $X{=}\{\}$ *then* σ
else Abs-memory (*fun-upd-equivp* $eq'\ \sigma'$ (*SOME* $x.\ x{\in}X$) *None*, *eq*))

lemma *reset-mt* : $\sigma\ (reset\ \{\}) = \sigma$
unfolding *reset-def Let-def*
by *simp*

lemma *reset-sh* :
assumes $*$: $(x\ shares_{\sigma}\ y)$
and $**$: $x \in X$
shows $\sigma\ (reset\ X)\ \$\ y = None$
oops

B.6 Memory Domain Definition

definition *Domain* :: $('\alpha, '\beta) memory \Rightarrow '\alpha\ set$
where $Domain\ \sigma = dom\ (fst\ (Rep\text{-}memory\ \sigma))$

B.7 Properties on Memory Domain

lemma *Domain-charn*:
assumes *1*:$x \in Domain\ \sigma$
shows $\exists\ y.\ Some\ y = fst\ (Rep\text{-}memory\ \sigma)\ x$
using *1*
by(*auto simp*: *Domain-def*)

lemma *Domain-charn1*:
assumes *1*:$x \in$ *Domain* σ
shows $\exists$ *y*. *the* (*Some y*) = σ \$ *x*
using *1*
by(*auto simp*: *Domain-def lookup-def*)

— This lemma says that if x and y are quivalent this means that they are in the same set of equivalent classes

lemma *shares-dom* [*code-unfold*, *intro*]:
assumes *x* $shares_\sigma$ *y*
shows (*x* $\in$ *Domain* σ) = (*y* $\in$ *Domain* σ)
using *insert Rep-memory*[*of* σ] *assms*
by (*auto simp*: *sharing-def Domain-def*)

lemma *Domain-mono*:
assumes *1*: *x* $\in$ *Domain* σ
and *2*: (*x* $shares_\sigma$ *y*)
shows *y* $\in$ *Domain* σ
using *1 2 Rep-memory*[*of* σ]
by (*auto simp add*: *sharing-def Domain-def*)

corollary *Domain-nonshares* :
assumes *1*: *x* $\in$ *Domain* σ
and *2*: *y* $\notin$ *Domain* σ
shows $\neg$(*x* $shares_\sigma$ *y*)
using *1 2 Domain-mono*
by(*fast*)

lemma *Domain-init*[*simp*] : *Domain init* = {}
unfolding *init-def Domain-def*
by(*simp-all add*:*identity-equivp Abs-memory-inverse*)

lemma *Domain-update*[*simp*] :*Domain* (σ (*x* :=\$ *y*)) = (*Domain* σ) $\cup$ {*y* . *y* $shares_\sigma$ *x*}
unfolding *update-def Domain-def sharing-def*
proof (*simp-all*)
have $*$: *Pair-upd-lifter* (*Rep-memory* σ) *x y* $\in$ {(σ, *R*). *equivp R* $\wedge$ ($\forall$ *x y*. *R x y* $\longrightarrow$ σ *x* = σ *y*)}
by (*simp*, *metis* (*lifting*, *mono-tags*) *Rep-memory mem-Collect-eq update_sound'*)
have $**$: *snd* (*Rep-memory* σ) *x x*
by(*metis equivp-reflp sharing-charn2*)
show *dom* (*fst* (*Rep-memory* (*Abs-memory* (*Pair-upd-lifter* (*Rep-memory* σ) *x y*)))) =
dom (*fst* (*Rep-memory* σ)) $\cup$ {*y*. *snd* (*Rep-memory* σ) *y x*}

apply(*simp-all add*: *Abs-memory-inverse*[*OF* ∗])
apply(*subst surjective-pairing* [*of* (*Rep-memory* σ)])
apply(*subst Pair-upd-lifter.simps*, *simp*)
apply(*auto simp*: ∗∗ *fun-upd-equivp-def*)
done
qed

lemma *Domain-share1*:
assumes *1* : $a \in Domain\ \sigma$
and *2* : $b \in Domain\ \sigma$
shows $Domain\ (\sigma(a \bowtie b)) = Domain\ \sigma$
proof(*simp-all add*:*Set.set-eq-iff*, *tactic ALLGOALS* (*rtac* @{*thm allI*}))
fix *x*
have ∗∗∗: *transfer-rep* (*Rep-memory* σ) (*id a*) (*id b*) ∈ {(σ, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ σ *x* = σ *y*)}
by (*metis* (*lifting*, *mono-tags*) *Rep-memory transfer-rep-sound*)
show (*x* ∈ *Domain* (σ (*a* ⋈ *b*))) = (*x* ∈ *Domain* σ)
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def*
apply(*subst Abs-memory-inverse*[*OF* ∗∗∗])
apply(*insert 1 2*, *simp add*: *o-def transfer-rep-simp Domain-def*)
apply(*auto split*: *split-if split-if-asm*)
done
qed

lemma *Domain-share-tgt* : $a \in Domain\ \sigma \Longrightarrow b \in Domain\ (\sigma\ (a \bowtie b))$
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def id-def*
apply(*subst Abs-memory-inverse*[*OF transfer-rep-sound2*])
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def id-def*
apply(*simp add*: *o-def transfer-rep-simp Domain-def*)
by(*auto split*: *split-if*)

lemma *Domain-share2* :
assumes *1* : $a \in Domain\ \sigma$
and *2* : $b \notin Domain\ \sigma$
shows $Domain\ (\sigma(a \bowtie b)) = (Domain\ \sigma - \{x.\ x\ shares_\sigma\ b\} \cup \{b\})$
proof(*simp-all add*:*Set.set-eq-iff*, *auto*)
fix *x*
assume *3* : $x \in Domain\ (\sigma\ (a \bowtie b))$
and *4* : $x \neq b$
show $x \in Domain\ \sigma$
apply(*insert 3 4*)
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def id-def*
apply(*subst* (*asm*) *Abs-memory-inverse*[*OF transfer-rep-sound2*])
apply(*insert 1* , *simp add*: *o-def transfer-rep-simp Domain-def*)
apply(*auto split*: *split-if split-if-asm*)
done

```
next
   fix x
   assume 3 : x ∈ Domain (σ (a ⋈ b))
    and  4 : x ≠ b
    and  5 : x shares_σ b
    have ** : x ∉ Domain σ  using 2 5 Domain-mono by (fast )
   show False
       apply(insert 3 4 5, erule contrapos-pp, simp)
       unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
       apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
       apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
       apply(auto split: split-if split-if-asm )
       using ** Domain-def domI apply fast
       done
next
   show b ∈ Domain (σ (a ⋈ b))
       using 1 Domain-share-tgt  by fast
next
   fix x
    assume 3 : x ∈ Domain σ
      and 4 : ¬ x shares_σ b
    show       x ∈ Domain (σ (a ⋈ b))
       unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
       apply(subst Abs-memory-inverse[OF transfer-rep-sound2])
       apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
       apply(auto split: split-if split-if-asm )
       using 3 Domain-def domD
       apply fast
       done
qed
```

```
lemma Domain-share3:
assumes 1 : a ∉  Domain σ
shows     Domain (σ(a⋈b)) = (Domain σ − {b})
proof(simp-all add:Set.set-eq-iff, auto)
 fix x
 assume 3: x ∈ Domain (σ (a ⋈ b))
 show      x ∈ Domain σ
    apply(insert 3)
    unfolding sharing-def Domain-def transfer-def map-fun-def o-def id-def
    apply(subst (asm) Abs-memory-inverse[OF transfer-rep-sound2])
    apply(insert 1 , simp add: o-def transfer-rep-simp Domain-def )
    apply(auto split: split-if split-if-asm )
    done
next
 assume 3: b ∈ Domain (σ (a ⋈ b))
```

show *False*
apply(*insert 1 3*)
apply(*erule contrapos-pp*[*of* $b \in$ *Domain* (σ ($a \bowtie b$))], *simp*)
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def id-def*
apply(*subst Abs-memory-inverse*[*OF transfer-rep-sound2*])
apply(*insert 1* , *simp add*: *o-def transfer-rep-simp Domain-def*)
apply(*auto split*: *split-if*)
done
next
fix *x*
assume *3*: $x \in$ *Domain* σ
and *4*: $x \neq b$
show $x \in$ *Domain* (σ ($a \bowtie b$))
apply(*insert 3 4*)
unfolding *sharing-def Domain-def transfer-def map-fun-def o-def id-def*
apply(*subst Abs-memory-inverse*[*OF transfer-rep-sound2*])
apply(*insert 1* , *simp add*: *o-def transfer-rep-simp Domain-def*)
apply(*auto split*: *split-if split-if-asm*)
done
qed

lemma *Domain-transfer* :
Domain ($\sigma(a \bowtie b)$) = (*if* $a \notin$ *Domain* σ
then (*Domain* $\sigma - \{b\}$)
else if $b \notin$ *Domain* σ
then (*Domain* $\sigma - \{x.\ x\ shares_\sigma\ b\} \cup \{b\}$)
else *Domain* σ)
using *Domain-share1 Domain-share2 Domain-share3*
by *metis*

lemma *Domain-transfer-approx* : *Domain* ($\sigma(a \bowtie b)$) $\subseteq$ *Domain* (σ) $\cup \{b\}$
by(*auto simp*: *Domain-transfer*)

B.8 Sharing Relation and Memory Update

lemma *sharing-upd*: $x\ shares_{(\sigma(a :=_\$ b))}\ y = x\ shares_\sigma\ y$
using *insert Rep-memory*[*of* σ]
by(*auto simp*: *sharing-def update-def Abs-memory-inverse*[*OF update_sound*])

— this lemma says that if we do an update on an adress x all the elements that are equivalent of x are updated

lemma *update''*:
σ ($x :=_\$ y$) = *Abs-memory*(*fun-upd-equivp* ($\lambda x\ y.\ x\ shares_\sigma\ y$) (*fst* (*Rep-memory* σ)) x (*Some* y),
snd (*Rep-memory* σ))
unfolding *update-def sharing-def*

by (*metis update′ update-def*)

theorem *update-cancel*:
assumes $x\ shares_\sigma\ x'$
shows $\sigma(x :=_\$ y)(x' :=_\$ z) = (\sigma(x' :=_\$ z))$
proof −
have $*$: (*fun-upd-equivp*(*snd*(*Rep-memory* σ))(*fst*(*Rep-memory* σ)) x (*Some* y),*snd* (*Rep-memory* σ))
$\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
unfolding *fun-upd-equivp-def*
by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
have $**$: $\bigwedge R\ \sigma.\ equivp\ R \Longrightarrow R\ x\ x' \Longrightarrow$
fun-upd-equivp R (*fun-upd-equivp R* σ x (*Some* y)) x' (*Some* z)
$=$ *fun-upd-equivp R* σ x' (*Some* z)
unfolding *fun-upd-equivp-def*
apply(*rule ext*)
apply(*case-tac R xa x′*, *auto*)
apply(*erule contrapos-np*, *erule equivp-transp*, *simp-all*)
done
show *?thesis*
apply(*simp add*: *update′*)
apply(*insert sharing-charn assms*[*simplified sharing-def*])
apply(*simp add*: *Abs-memory-inverse* [*OF* $*$] $**$)
done
qed

theorem *update-commute*:
assumes *1*:$\neg$ ($x\ shares_\sigma\ x'$)
shows $(\sigma(x :=_\$ y))(x' :=_\$ z) = (\sigma(x' :=_\$ z)(x :=_\$ y))$
proof −
have $*$: $\bigwedge x\ y$.(*fun-upd-equivp*(*snd*(*Rep-memory* σ))(*fst*(*Rep-memory* σ)) x (*Some* y),*snd* (*Rep-memory* σ))
$\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
unfolding *fun-upd-equivp-def*
by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
have $**$: $\bigwedge R\ \sigma.\ equivp\ R \Longrightarrow \neg\ R\ x\ x' \Longrightarrow$
fun-upd-equivp R (*fun-upd-equivp R* σ x (*Some* y)) x' (*Some* z) $=$
fun-upd-equivp R (*fun-upd-equivp R* σ x' (*Some* z)) x (*Some* y)
unfolding *fun-upd-equivp-def*
apply(*rule ext*)
apply(*case-tac R xa x′*, *auto*)
apply(*erule contrapos-np*)
apply(*frule equivp-transp*, *simp-all*)
apply(*erule equivp-symp*, *simp-all*)
done
show *?thesis*
apply(*simp add*: *update′*)
apply(*insert assms*[*simplified sharing-def*])

apply(*simp add*: *Abs-memory-inverse* [*OF* ∗] ∗∗)
done
qed

B.9 Properties on lookup and update wrt the Sharing Relation

lemma *update-triv*:
assumes *1*: $x\ shares_\sigma\ y$
and *2*: $y \in Domain\ \sigma$
shows $\sigma\ (x :=_\$ (\sigma\ \$\ y)) = \sigma$
proof −
{
fix *z*
assume *zx*: $z\ shares_\sigma\ x$
then have *zy*: $z\ shares_\sigma\ y$
using *1* **by** (*rule sharing-trans*)
have *F*: $y \in Domain\ \sigma \Longrightarrow x\ shares_\sigma\ y$
$\Longrightarrow Some\ (the\ (fst\ (Rep\text{-}memory\ \sigma)\ x)) = fst\ (Rep\text{-}memory\ \sigma)\ y$
by(*auto simp*: *Domain-def dest*: *shares-result*)
have $Some\ (the\ (fst\ (Rep\text{-}memory\ \sigma)\ y)) = fst\ (Rep\text{-}memory\ \sigma)\ z$
using *zx* **and** *shares-result* [*OF zy*] *shares-result* [*OF zx*]
using *F* [*OF 2 1*]
by *simp*
} **note** *3* = *this*
show *?thesis*
unfolding *update′′ lookup-def fun-upd-equivp-def*
by (*simp add*: *3 Rep-memory-inverse if-cong*)
qed

lemma *update-idem′* :
assumes *1*: $x\ shares_\sigma\ y$
and *2*: $x \in Domain\ \sigma$
and *3*: $\sigma\ \$\ x = z$
shows $\sigma(y:=_\$ z) = \sigma$
proof −
have ∗ : $y \in Domain\ \sigma$
by(*simp add*: *shares-dom*[*OF 1*, *symmetric*] *2*)
have ∗∗: $\sigma\ (x :=_\$ (\sigma\ \$\ y)) = \sigma$
using *1 2* ∗
by (*simp add*: *update-triv*)
also have $(\sigma\ \$\ y) = \sigma\ \$\ x$
by (*simp only*: *lookup-def shares-result* [*OF 1*])
finally show *?thesis*
using *1 2 3 sharing-sym update-triv*
by *fast*
qed

lemma *update-idem* :

assumes *2*: $x \in Domain\ \sigma$
and *3*: $\sigma\ \$\ x = z$
shows $\sigma(x{:=}_\$\ z) = \sigma$
proof −
show *?thesis*
using *2 3 sharing-refl update-triv*
by *fast*
qed

lemma *update-apply*: $(\sigma(x :=_\$ y))\ \$\ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma\ \$\ z)$
proof −
have $*$: $(\lambda z.\ if\ z\ shares_\sigma\ x\ then\ Some\ y\ else\ fst\ (Rep\text{-}memory\ \sigma)\ z,\ snd\ (Rep\text{-}memory\ \sigma))$
$\in \{(\sigma, R).\ equivp\ R \wedge (\forall x\ y.\ R\ x\ y \longrightarrow \sigma\ x = \sigma\ y)\}$
unfolding *sharing-def*
by(*rule update_sound*[*simplified fun-upd-equivp-def*], *simp*)
show *?thesis*
proof (*cases z shares*$_\sigma$ *x*)
case *True*
assume *A*: *z shares*$_\sigma$ *x*
show $\sigma\ (x :=_\$ y)\ \$\ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma\ \$\ z)$
unfolding *update'' lookup-def fun-upd-equivp-def*
by(*simp add*: *Abs-memory-inverse* [*OF* $*$])
next
case *False*
assume *A*: ¬ *z shares*$_\sigma$ *x*
show $\sigma\ (x :=_\$ y)\ \$\ z = (if\ z\ shares_\sigma\ x\ then\ y\ else\ \sigma\ \$\ z)$
unfolding *update'' lookup-def fun-upd-equivp-def*
by(*simp add*: *Abs-memory-inverse* [*OF* $*$])
qed
qed

lemma *update-share*:
assumes *z shares*$_\sigma$ *x*
shows $\sigma(x :=_\$ a)\ \$\ z = a$
using *assms*
by (*simp only*: *update-apply if-True*)

lemma *update-other*:
assumes ¬(*z shares*$_\sigma$ *x*)
shows $\sigma(x :=_\$ a)\ \$\ z = \sigma\ \$\ z$
using *assms*
by (*simp only*: *update-apply if-False*)

lemma *lookup-update-rep*:
assumes *1*: (*snd* (*Rep-memory* σ')) *x y*
shows (*fst* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)) *x* =
(*fst* (*Pair-upd-lifter* (*Rep-memory* σ') *src dst*)) *y*
using *1 shares-result sharing-def sharing-upd update.rep-eq*

by (*metis* (*hide-lams*, *no-types*))

lemma *lookup-update-rep''*:
assumes *1*: *x shares*$_\sigma$ *y*
shows (*σ* (*src* :=$_\$$ *dst*)) $ *x* = (*σ* (*src* :=$_\$$ *dst*)) $ *y*
using *1 lookup-def lookup-update-rep sharing-def update.rep-eq*
by *metis*

theorem *memory-ext* :
assumes ∗ : ⋀ *x y*. (*x shares*$_\sigma$ *y*) = (*x shares*$_{\sigma'}$ *y*)
and ∗∗ : *Domain σ* = *Domain σ'*
and ∗∗∗ : ⋀ *x*. *σ* $ *x* = *σ'* $ *x*
shows *σ* = *σ'*
apply(*subst Rep-memory-inverse*[*symmetric*])
apply(*subst* (*3*) *Rep-memory-inverse*[*symmetric*])
apply(*rule arg-cong*[*of - - Abs-memory*])
apply(*auto simp*:*Product-Type.prod-eq-iff*)
proof −
show *fst* (*Rep-memory σ*) = *fst* (*Rep-memory σ'*)
apply(*rule ext*, *insert* ∗∗ ∗∗∗, *simp add*: *lookup-def Domain-def*)
apply (*metis* (*lifting*, *no-types*) *domD domIff the.simps*)
done
next
show *snd* (*Rep-memory σ*) = *snd* (*Rep-memory σ'*)
by(*rule ext*, *rule ext*, *insert* ∗, *simp add*: *sharing-def*)
qed

Nice connection between sharing relation, domain of the memory and content equaltiy on the one hand and equality on the other; this proves that our memory model is fully abstract in these three operations.

corollary *memory-ext2*: (*σ* = *σ'*) = ((∀ *x y*. (*x shares*$_\sigma$ *y*) = (*x shares*$_{\sigma'}$ *y*))
∧ *Domain σ* = *Domain σ'*
∧ (∀ *x*. *σ* $ *x* = *σ'* $ *x*))
by(*auto intro*: *memory-ext*)

B.10 Rules On Sharing and Memory Transfer

lemma *transfer-rep-inv-E*:
assumes *1* : *σ* ∈ {(*σ*, *R*). *equivp R* ∧ (∀ *x y*. *R x y* ⟶ *σ x* = *σ y*)}
and *2* : *memory-inv* (*transfer-rep σ src dst*) ⟹ *Q*
shows *Q*
using *assms transfer-rep-sound*[*of σ*]
by (*auto simp*: *Abs-memory-inverse*)

lemma *transfer-rep-fst1*:
assumes *1*: *σ* = *fst*(*transfer-rep* (*Rep-memory σ'*) *src dst*)

shows $\bigwedge x.\ x = dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ src$
using *1* **unfolding** *transfer-rep-simp*
by *simp*

lemma *transfer-rep-fst2*:
assumes *1*: $\sigma = fst(transfer\text{-}rep\ \ (Rep\text{-}memory\ \ \sigma')\ src\ dst)$
shows $\bigwedge x.\ x \neq\ dst \Longrightarrow \sigma\ x = (fst\ (Rep\text{-}memory\ \ \sigma'))\ (id\ x)$
using *1* **unfolding** *transfer-rep-simp*
by *simp*

lemma *lookup-transfer-rep'*:
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ src =$
$(fst\ (transfer\text{-}rep\ (Rep\text{-}memory\ \ \sigma')\ src\ dst))\ dst$
using *Rep-memory* [*of* σ']
apply (*erule-tac src*= *src* **and** *dst* = *dst* **in** *transfer-rep-inv-E*)
apply (*rotate-tac 1*)
apply (*subst* (*asm*) *surjective-pairing*[*of* (*transfer-rep* (*Rep-memory* σ') *src dst*)])
unfolding *memory-inv.simps*
apply (*erule conjE*)
apply (*erule allE*)+
apply (*erule impE*)
unfolding *transfer-rep-simp*
apply *auto*
using *equivp-reflp snd-memory-equivp*
apply *metis*
done

theorem *share-transfer*:
$x\ shares_{\sigma(a \bowtie b)}\ y = (\ (y = b \wedge (x = b \vee (x \neq b \wedge\ x\ shares_\sigma\ a))) \vee (y \neq b \wedge ((x = b \wedge a\ shares_\sigma\ y) \vee (x \neq b \wedge x\ shares_\sigma\ y))))$
unfolding *sharing-def transfer-def*
unfolding *transfer-def map-fun-def o-def id-def*
apply(*subst Abs-memory-inverse*[*OF transfer-rep-sound2*], *simp add*: *transfer-rep-simp*)
by (*metis equivp-reflp sharing-charn2*)

lemma *transfer-share*:$a\ shares_{\sigma(a \bowtie b)}\ b$ **by**(*simp add*: *share-transfer sharing-refl*)

lemma *transfer-share-sym*:$a\ shares_{\sigma\ (b \bowtie a)}\ b$ **by**(*simp add*: *share-transfer sharing-refl*)

lemma *transfer-share-mono*:$x\ shares_\sigma\ y \Longrightarrow \neg(x\ shares_\sigma\ b) \Longrightarrow (x\ shares_{\sigma\ (a \bowtie b)}\ y)$
by(*auto simp*: *share-transfer sharing-refl*)

lemma *transfer-share-charn*:
$\neg(x\ shares_\sigma\ b) \Longrightarrow \neg(y\ shares_\sigma\ b) \Longrightarrow x\ shares_{\sigma(a \bowtie b)}\ y = x\ shares_\sigma\ y$
by(*auto simp*: *share-transfer sharing-refl*)

lemma *transfer-share-trans*:$(a\ shares_\sigma\ x) \Longrightarrow (x\ shares_{\sigma(a \bowtie b)}\ b)$
by(*auto simp*: *share-transfer sharing-refl sharing-sym*)

lemma *transfer-share-trans-sym*:$(a\ shares_\sigma\ y) \Longrightarrow (b\ shares_{(\sigma(a \bowtie b))}\ y)$
using *transfer-share-trans sharing-sym* **by** *fast*

lemma *transfer-share-trans'*: $(a\ shares_{(\sigma(a \bowtie b))}\ z) \Longrightarrow (b\ shares_{(\sigma(a \bowtie b))}\ z)$
using *transfer-share sharing-sym sharing-trans* **by** *fast*

lemma *transfer-tri* : $x\ shares_{\sigma\ (a \bowtie b)}\ y \Longrightarrow x\ shares_\sigma\ b \vee b\ shares_\sigma\ y \vee x\ shares_\sigma\ y$
by (*metis sharing-sym transfer-share-charn*)

lemma *transfer-tri'* : $\neg\ x\ shares_{\sigma\ (a \bowtie b)}\ y \Longrightarrow\ y\ shares_\sigma\ b \vee \neg\ x\ shares_\sigma\ y$
by (*metis sharing-sym sharing-trans transfer-share-mono*)

lemma *transfer-dest'* :
assumes *1* : $a\ shares_{\sigma\ (a \bowtie b)}\ y$
and *2* : $b \neq y$
shows $a\ shares_\sigma\ y$
using *assms* **by**(*auto simp*: *share-transfer sharing-refl sharing-sym*)

lemma *transfer-dest* :
assumes *1* : $\neg(x\ shares_\sigma\ a)$
and *2* : $x \neq b$
and *3* : $x\ shares_\sigma\ b$
shows $\neg(x\ shares_{\sigma\ (a \bowtie b)}\ b)$
using *assms* **by**(*auto simp*: *share-transfer sharing-refl sharing-sym*)

lemma *transfer-dest''*:$x = b \Longrightarrow y\ shares_\sigma\ a \Longrightarrow x\ shares_{\sigma(a \bowtie b)}\ y$
by (*metis sharing-sym transfer-share-trans-sym*)

thm *share-transfer*
transfer-share
transfer-share-sym

sharing-sym [*THEN transfer-share-trans*]

sharing-sym [*THEN transfer-share-trans-sym*]

transfer-share-trans′
transfer-dest′′
transfer-dest′
transfer-tri′
transfer-share-mono
transfer-tri
transfer-share-charn
transfer-dest

B.11 Properties on Memory Transfer and Lookup

lemma *transfer-share-lookup1*: $(\sigma(x \bowtie y)) \$ x = \sigma \$ x$
using *lookup-transfer-rep′ transfer-rep-fst1*
unfolding *lookup-def transfer.rep-eq*
by *metis*

lemma *transfer-share-lookup2*:
$(\sigma(x \bowtie y)) \$ y = \sigma \$ x$
using *transfer-rep-fst1*
unfolding *transfer.rep-eq lookup-def*
by *metis*

lemma add_e*-not-share-lookup*:
assumes *1*: $\neg(x\ shares_\sigma\ z)$
and *2*: $\neg(y\ shares_\sigma\ z)$
shows $\sigma\ (x \bowtie y) \$ z = \sigma \$ z$
using *assms*
unfolding *sharing-def lookup-def transfer.rep-eq*
using *id-def sharing-def sharing-refl transfer-rep-fst2*
by *metis*

lemma *transfer-share-dom*:
assumes *1*: $z \in Domain\ \sigma$
and *2*: $\neg(y\ shares_\sigma\ z)$
shows $(\sigma(x \bowtie y)) \$ z = \sigma \$ z$
using *assms*
unfolding *Domain-def sharing-def lookup-def*
using *2 transfer.rep-eq id-apply sharing-refl transfer-rep-fst2*
by *metis*

lemma *shares-result′*:
assumes *1*: $(x\ shares_\sigma\ y)$
shows $\sigma \$ x = \sigma \$ y$
using *assms lookup-def shares-result*
by *metis*

lemma *transfer-share-cancel1*:
assumes *1*: $(x\ shares_\sigma\ z)$
shows $(\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ x$
using *1 transfer.rep-eq transfer-share-trans lookup-def transfer-rep-fst1 shares-result*
by (*metis*)

B.12 Test on Sharing and Transfer via smt ...

lemma $\forall x\ y.\ x \neq y \longrightarrow \neg(x\ shares_\sigma\ y) \Longrightarrow \sigma\ \$\ x > \sigma\ \$\ y \Longrightarrow \sigma(3 \bowtie (4::nat)) = \sigma' \Longrightarrow \sigma'' = (\sigma'(3 :=_\$ ((\sigma'\ \$\ 4) + 2))) \Longrightarrow x \neq 3 \Longrightarrow x \neq 4 \Longrightarrow y \neq 3 \Longrightarrow y \neq 4 \Longrightarrow \sigma''\ \$\ x > \sigma''\ \$\ y$
by (*smt add$_e$-not-share-lookup transfer-share-charn update-apply*)

B.13 Instrumentation of the smt Solver

lemma *transfer-share-charn-smt* :
$\neg(i\ \ shares_\sigma\ k') \wedge \neg(k\ shares_\sigma\ k') \longrightarrow i\ shares_{\sigma(i' \bowtie k')}\ k = i\ shares_\sigma\ k$
using *transfer-share-charn*
by *fast*

lemma *add$_e$-not-share-lookup-smt*:
$\neg(x\ shares_\sigma\ z) \wedge \neg(y\ shares_\sigma\ z) \longrightarrow (\sigma\ (x \bowtie y)\ \$\ z) = (\sigma\ \$\ z)$
using *add$_e$-not-share-lookup*
by *auto*

lemma *transfer-share-dom-smt*:
$z \in Domain\ \sigma \wedge \neg(y\ shares_\sigma\ z) \longrightarrow (\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ z$
using *transfer-share-dom*
by *auto*

lemma *transfer-share-cancel1-smt*:
$(x\ shares_\sigma\ z) \longrightarrow (\sigma(x \bowtie y))\ \$\ z = \sigma\ \$\ x$
using *transfer-share-cancel1*
by *auto*

lemma *lookup-update-rep''-smt*:
$x\ shares_\sigma\ y \longrightarrow (\sigma\ (src :=_\$ dst))\ \$\ x = (\sigma\ (src :=_\$ dst))\ \$\ y$
using *lookup-update-rep''*
by *auto*

theorem *update-commute-smt*:
$\neg\ (x\ shares_\sigma\ x') \longrightarrow ((\sigma(x :=_\$ y))(x' :=_\$ z)) = (\sigma(x' :=_\$ z)(x :=_\$ y))$

using *update-commute*
by *auto*

theorem *update-cancel-smt*:
$(x\ shares_\sigma\ x') \longrightarrow (\sigma(x :=_\$\ y)(x' :=_\$\ z)) = (\sigma(x' :=_\$\ z))$
using *update-cancel*
by *auto*

lemma *update-other-smt*:
$\neg(z\ shares_\sigma\ x) \longrightarrow (\sigma(x :=_\$\ a)\ \$\ z) = \sigma\ \$\ z$
using *update-other*
by *auto*

lemma *update-share-smt*:
$(z\ shares_\sigma\ x) \longrightarrow (\sigma(x :=_\$\ a)\ \$\ z) = a$
using *update-share*
by *auto*

lemma *update-idem-smt* :
$(x\ shares_\sigma\ y) \wedge x \in Domain\ \sigma \wedge (\sigma\ \$\ x = z) \longrightarrow (\sigma(x{:=_\$}\ z)) = \sigma$
using *update-idem*
by *fast*

lemma *update-triv-smt*:
$(x\ shares_\sigma\ y) \wedge y \in Domain\ \sigma \longrightarrow (\sigma\ (x :=_\$\ (\sigma\ \$\ y))) = \sigma$
using *update-triv*
by *auto*

lemma *shares-result-smt*:
$x\ shares_\sigma\ y \longrightarrow \sigma\ \$\ x = \sigma\ \$\ y$
using *shares-result'*
by *fast*

lemma *shares-dom-smt* :
$x\ shares_\sigma\ y \longrightarrow (x \in Domain\ \sigma) = (y \in Domain\ \sigma)$
using *shares-dom* **by** *fast*

lemma *sharing-sym-smt* :
$x\ shares_\sigma\ y \longrightarrow y\ shares_\sigma\ x$
using *sharing-sym*
by (*auto*)

lemma *sharing-trans-smt*:
$x\ shares_\sigma\ y \wedge y\ shares_\sigma\ z \longrightarrow x\ shares_\sigma\ z$
using *sharing-trans*

by(*auto*)

lemma *nat-0-le-smt*: $0 \leq z \longrightarrow int\ (nat\ z) = z$
by *transfer clarsimp*

lemma *nat-le-0-smt*: $0 > z \longrightarrow int\ (nat\ z) = 0$
by *transfer clarsimp*

lemma *transfer-share-trans-smt*:
$(x\ shares_\sigma\ z) \longrightarrow (z\ shares_{\sigma(x \bowtie y)}\ y)$
using *transfer-share-trans*
by *fast*

lemma *transfer-share-mono-smt*:
$(x\ shares_\sigma\ y) \wedge \neg(x\ shares_\sigma\ y') \longrightarrow (x\ shares_{\sigma\ (x' \bowtie y')}\ y)$
using *transfer-share-mono*
by *fast*

lemma *transfer-share-trans'-smt*:
$(x\ shares_{(\sigma(x \bowtie y))}\ z) \longrightarrow (y\ shares_{(\sigma(x \bowtie y))}\ z)$
using *transfer-share-trans'*
by *fast*

lemma *transfer-share-old-new-trans-smt*:
$(x\ shares_\sigma\ z) \longrightarrow (y\ shares_{(\sigma(x \bowtie y))}\ z)$
using *transfer-share-trans-sym*
by *fast*

lemma *transfer-share-old-new-trans1-smt*:
$a\ shares_\sigma\ b \wedge a\ shares_\sigma\ c \longrightarrow (c\ shares_{(\sigma\ (a \bowtie d))}\ b\ )$
using *transfer-share-trans-smt sharing-sym-smt sharing-trans-smt*
by *metis*

lemma *Domain-mono-smt*:
$x \in Domain\ \sigma \wedge (x\ shares_\sigma\ y) \longrightarrow y \in Domain\ \sigma$
using *Domain-mono*
by *fast*

lemma *sharing-upd-smt*: $x\ shares_{(\sigma(a :=_\$ b))}\ y = x\ shares_\sigma\ y$
using *sharing-upd*
by *fast*

lemma *sharing-init-mem-list-smt* :
i ≠ *k* ⟶ ¬(*i* *shares*$_{init\text{-}mem\text{-}list\ S}$ *k*)
using *sharing-init-mem-list*
by *fast*

lemma *mem1-smt*:(σ(*a*⋈*b*) $ *a*) = (σ(*a*⋈*b*) $ *b*)
by (*metis transfer-share-lookup1 transfer-share-lookup2*)

B.14 Tools for the initialization of the memory

definition *memory-fst-eq*$_{init}$ :: *int list* ⇒ *int list* ⇒ (*int*, *int*)*memory*
where *memory-fst-eq*$_{init}$ *ADD VAL* =
(*foldl* (λ *m* (*x*, *y*). (*m* (*x*:=$_{\$}$*y*))) *init* (*zip ADD VAL*))

definition *memory-snd-eq*$_{init}$ :: *int list* ⇒ *int list* ⇒(*int*, *int*)*memory* ⇒(*int*, *int*)*memory*
where *memory-snd-eq*$_{init}$ *SRC DST m* =
(*foldl* (λ*m* (*x*, *y*). (*m* (*x*⋈*y*))) *m* (*zip SRC DST*))

definition *memory-eq*$_{init}$ :: *int list* ⇒ *int list* ⇒ *int list* ⇒(*int*, *int*)*memory*
where *memory-eq*$_{init}$ *SRC VAL DST* =
foldl (λ *m* (*SRC*, *DST*). *memory-snd-eq*$_{init}$ *SRC DST m*)
(*memory-fst-eq*$_{init}$ *SRC VAL*) [(*SRC*, *DST*)]

lemmas *sharing-smt* = *sharing-refl* *transfer-share*
sharing-commute *nat-le-0-smt*
nat-0-le-smt *sharing-sym-smt*
transfer-share-lookup1 *transfer-share-lookup2*
sharing-init-mem-list-smt *sharing-upd-smt*
shares-result-smt *transfer-share-old-new-trans-smt*
transfer-share-trans-smt *mem1-smt*
update-share-smt *shares-dom-smt*
Domain-mono-smt *sharing-trans-smt*
transfer-share-cancel1-smt *transfer-share-trans'-smt*
update-apply *update-other-smt*
update-cancel-smt *transfer-share-old-new-trans1-smt*
lookup-update-rep''-smt *update-triv-smt*
transfer-share-mono-smt *update-commute-smt*
transfer-share-dom-smt *add*$_e$*-not-share-lookup-smt*
update-idem-smt *transfer-share-charn-smt*

lemmas *sharing-refl-smt* = *sharing-refl*

B.15 An Intrastructure for Global Memory Spaces

Memory spaces are common concepts in Operating System (OS) design since it is a major objective of OS kernels to separate logical, linear memory spaces belonging to different processes (or in other terminologies such as PiKeOS: tasks) from each other. We achieve this goal by modeling the adresses of memory spaces as a *pair* of a subject (e.g. process or task, denominated by a process-id or task-id) and a location (a conventional adress).

Our model is still generic - we do not impose a particular type for subjects or locations (which could be modeled in a concrete context by an enumeration type as well as integers of bitvector representations); for the latter, however, we require that they are instances of the type class $'\alpha$ assuring that there is a minimum of infrastructure for address calculation: there must exist a $0::'a$-element, a distinct $1::'a$-element and an addition operation with the usual properties.

fun $init_{globalmem}$ :: $(('sub \times 'loc :: comm\text{-}semiring\text{-}1), '\beta)$ *memory*
$\Rightarrow$ $('sub \times 'loc) \Rightarrow '\beta$ *list*
$\Rightarrow (('sub \times 'loc), '\beta)$ *memory* (- |> - <| - [60,60,60] 70)
where σ |> *start* <| [] = σ
| σ |> (*sub*,*loc*) <| (*a* # *S*) = ((σ((*sub*,*loc*):=$_\$$ *a*)) |> (*sub*, *loc*+1)<| *S*)

lemma *Domain-mem-init-Nil* : *Domain*(σ |> *start* <| []) = *Domain* σ
by *simp*

Example

type-synonym *task-id* = *int*
type-synonym *loc* = *int*

type-synonym *global-mem* = ((*task-id*×*loc*), *int*)*memory*

definition σ_0 :: *global-mem*
where $\sigma_0 \equiv$ *init* |> (0,0) <| [0,0,0,0]
|> (2,0) <| [0,0]
|> (4,0) <| [2,0]

lemma σ_0-*Domain*: *Domain* σ_0 = {(4, 1), (4, 0), (2, 1), (2, 0), (0, 3), (0, 2), (0, 1), (0, 0)}
unfolding σ_0-*def*
by(*simp add*: *sharing-upd*)

notation *transfer* (add_e)
lemmas add_e-*def* = *transfer-def*
lemmas add_e-*rep-eq* = *transfer.rep-eq*
lemmas *transfer-share-old-new-trans* = *transfer-share-trans-sym*

lemmas *sharing-commute-smt = sharing-commute*
lemmas *update-apply-smt = update-apply*
lemmas *transfer-share-lookup2-smt = transfer-share-lookup2*
lemmas *transfer-share-lookup1-smt = transfer-share-lookup1*
lemmas *transfer-share-smt = transfer-share*

end

theory *IPC-errors-type*
imports *../TypeSchemes*
../Memory/SharedMemoryNew

begin

B.16 Error codes datatype

C HOL representation of PikeOS IPC error codes

— error codes are returned if an IPC action is aborted, the error codes has the following specificities:

- Must indicates which stage the error was occured.
- Each IPC stage has its own set of error codes
- Errors in the receiving stages does not affect sending stages
- Errors in sending stages affect receiving stages

We have another type of errors which is related to the different memory functionality.

— IPC errors
datatype *error-IPC =*

no-IPC-error
| *error-IPC-4* — if an action is used in stepping function with the wrong stage
— errors of the SEND part of IPC

| *error-IPC-21-in-PREP-SEND* — IF the receiver is an OR
| *error-IPC-22-in-PREP-SEND* — IF the receiver is an CR and the sender is not the one who can send msg to this receiver
| *error-IPC-23-in-PREP-SEND* — IF the receiver is an NR
| *error-IPC-4-in-PREP-SEND*— if an action is used in the wrong stage

| *error-IPC-21-in-PREP-RECV* — IF the receiver is an OR
| *error-IPC-22-in-PREP-RECV* — IF the receiver is an CR and the sender is not the one who can send msg to this receiver
| *error-IPC-23-in-PREP-RECV* — IF the receiver is an NR
| *error-IPC-4-in-PREP-RECV*— if an action is used in the wrong stage

| *error-IPC-1-in-WAIT-SEND* — if the thread has no rights to communicate with his partner
| *error-IPC-2-in-WAIT-SEND* — if the thread has no rights to access to this list of virtual adresses
| *error-IPC-3-in-WAIT-SEND* — if the thread try to send an IPC msg to him self
| *error-IPC-4-in-WAIT-SEND*— if an action is used in the wrong stage
| *error-IPC-5-in-WAIT-SEND* — if the receiver dont exist in the list of threads in the systeme
| *error-IPC-6-in-WAIT-SEND* — if the list of threads in the systeme is Nil
| *error-IPC-7-in-WAIT-SEND* — if the caller can not communicate with the receiver

|*error-IPC-1-in-BUF-SEND* — if the thread has no rights to access to this list of virtual adresses

|*error-IPC-1-in-BUF-RECV* — if the thread has no rights to access to this list of virtual adresses

| *error-IPC-1-in-WAIT-RECV* — if the thread has no rights to communicate with his partner
| *error-IPC-2-in-WAIT-RECV* — if the thread has no rights to access to this list of virtual adresses
| *error-IPC-3-in-WAIT-RECV* — if the thread try to send an IPC msg to him self
| *error-IPC-4-in-WAIT-RECV*— if an action is used in the wrong stage
| *error-IPC-5-in-WAIT-RECV* — if the receiver dont exist in the list of threads in the systeme Go to Done stage
| *error-IPC-6-in-WAIT-RECV* — if the list of threads in the systeme is Nil
| *error-IPC-7-in-WAIT-RECV* — if the caller can not communicate with the receiver

— memory errors
datatype *error-memory* =
no-mem-error — no errors related to memory adresses
| *not-valid-sender-addr-in-PREP-SEND* — error related to the adresses of the sender
| *not-valid-receiver-addr-in-PREP-SEND* — error related to the adresses of the receiver
| *not-valid-receiver-addr-in-PREP-RECV*
| *not-valid-sender-addr-in-PREP-RECV*

— datatype that contain memory and IPC errors

datatype *errors* =
NO-ERRORS
| *ERROR-MEM error-memory*
| *ERROR-IPC error-IPC*

type-synonym $error_{ipc}$ = *errors*
end

theory *IPC-thread-type*
imports *../Memory/SharedMemoryNew*
../TypeSchemes

begin

D HOL representation of PikeOS threads type

datatype *thread-state* = *CURRENT* | *WAITING* | *READY* | *STOPPED* | *INACTIVE*

In addition to the communication rights, the scope of IPC communication can further constrained by the receiving thread.

- If thread initiates an OR operation, any threads having rights can send msg to this thread.
- If thread initiates CR operation, it limits the IPC sending partner to one specific thread.
- If thread initiates NR operation, no thread can send a message to this thread.

datatype *th-ipc-st* =
OR — Open Receive
| *CR* — Close Receive
| *NR* — Nil Receive

datatype $partition_{enum}$ =
part0 |*part1* |*part2*

datatype $task_{enum}$ =
task0 |*task1* |*task2*

datatype $thread_{enum}$ =
th0 |*th1* |*th2*

type-synonym $thread_{id}$ = ($partition_{enum}$ * $task_{enum}$ * $thread_{enum}$)

type-synonym $thread_{ipc}$ = ($thread_{id}$, *thread-state*, *th-ipc-st*, (*int*, *int*) *memory* ,$thread_{id}$) *thread*

D.1 interface between thread and memory

definition *update-th-smm-equiv*

where *update-th-smm-equiv th addr val* = *update* (*own-vmem-adr th*) *addr val*

D.2 Relation between threads adresses and memory adresses

This section contains some predicate that defines relations between own thread addresses and memory addresses those predicate will be used to define some error codes related to own thread addresses.

— predicate that specify if this list of addresses are part of the addresses of the memory

definition *is-part-mem* ::
('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-mem mem addr* = (*addr* ∈ (*dom o fst o Rep-memory*) *mem*)

definition *is-part-mem-th* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-mem-th th mem addr* = (*is-part-mem* (*own-vmem-adr th*) *addr* ⟶ *is-part-mem mem addr*)

— predicate that specify if this list of addresses are part of the an other list of addresses

definition *is-part-addr-addr* ::
('*a*, '*b*) *memory* ⇒ ('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-addr-addr mem mem' addr*= (*is-part-mem mem' addr* ⟶ *is-part-mem mem addr*)

— This definition assures that a given list of addresses is part of list of addresses of thread
definition *is-part-addr-th* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒ '*a* ⇒ *bool*
where *is-part-addr-th th addr* = (*is-part-mem* (*own-vmem-adr th*) *addr*)

— This predicate assures that a given list of addresses is a part of memory addresses and part of thread addresses and the thread addresses are part of the memory

definition *is-part-addr-th-mem* ::
('*c*, '*d*, '*e*, ('*a*, '*b*) *memory*, '*f*, '*g*) *thread-scheme* ⇒ ('*a*, '*b*) *memory* ⇒ '*a* ⇒ *bool*
where *is-part-addr-th-mem th mem ns* = (*is-part-addr-addr mem* (*own-vmem-adr th*) *ns*)

lemma [*simp*]:*is-part-addr-th-mem th mem ns* = *is-part-mem-th th mem ns*
unfolding *is-part-addr-th-mem-def is-part-mem-th-def is-part-addr-addr-def*
by *simp*

D.3 Updating thread list in the state

— We will specify thread list inside our system by a partial function that takes a thread id and returns thread informations

type-synonym $('th\text{-}id, 'th\text{-}info)\ thread\text{-}tab = 'th\text{-}id \rightharpoonup 'th\text{-}info$

fun *thread-tab-update* ::
$('th\text{-}id \rightharpoonup 'th\text{-}info) \Rightarrow 'th\text{-}id \Rightarrow 'th\text{-}info \Rightarrow ('th\text{-}id \rightharpoonup 'th\text{-}info)$
where $thread\text{-}tab\text{-}update\ th\text{-}tab\ th\text{-}id\ th\text{-}info = th\text{-}tab(th\text{-}id \mapsto th\text{-}info)$

— Invariant on updating thread table

fun *update-th-waiting-true*::
$('th\text{-}id \rightharpoonup ('a, thread\text{-}state, 'b, 'c, 'd, 'e)\ thread\text{-}scheme) \Rightarrow 'th\text{-}id \Rightarrow bool$
where $update\text{-}th\text{-}waiting\text{-}true\ th\text{-}tab\ th\text{-}id =$
$(th\text{-}id \in dom\ th\text{-}tab \wedge ((th\text{-}state\ o\ the\ o\ th\text{-}tab)\ th\text{-}id) = WAITING)$

fun *update-th-ready-true*::
$('th\text{-}id \rightharpoonup ('a, thread\text{-}state, 'b, 'c, 'd, 'e)\ thread\text{-}scheme) \Rightarrow 'th\text{-}id \Rightarrow bool$
where $update\text{-}th\text{-}ready\text{-}true\ th\text{-}tab\ th\text{-}id =$
$(th\text{-}id \in dom\ th\text{-}tab \wedge ((th\text{-}state\ o\ the\ o\ th\text{-}tab)\ th\text{-}id) = READY)$

fun *update-th-current-true*::
$('th\text{-}id \rightharpoonup ('a, thread\text{-}state, 'b, 'c, 'd, 'e)\ thread\text{-}scheme) \Rightarrow 'th\text{-}id \Rightarrow bool$
where $update\text{-}th\text{-}current\text{-}true\ th\text{-}tab\ th\text{-}id =$
$(th\text{-}id \in dom\ th\text{-}tab \wedge ((th\text{-}state\ o\ the\ o\ th\text{-}tab)\ th\text{-}id) = CURRENT)$

fun *update-th-stopped-true*::
$('th\text{-}id \rightharpoonup ('a, thread\text{-}state, 'b, 'c, 'd, 'e)\ thread\text{-}scheme) \Rightarrow 'th\text{-}id \Rightarrow bool$
where $update\text{-}th\text{-}stopped\text{-}true\ th\text{-}tab\ th\text{-}id =$
$(th\text{-}id \in dom\ th\text{-}tab \wedge ((th\text{-}state\ o\ the\ o\ th\text{-}tab)\ th\text{-}id) = STOPPED)$

— update functions for thread state

fun *update-th-waiting*
where $update\text{-}th\text{-}waiting\ th\text{-}id\ th\text{-}tab\ = (if\ th\text{-}id \in dom\ th\text{-}tab$
$then\ th\text{-}tab(th\text{-}id \mapsto ((the\ o\ th\text{-}tab)\ th\text{-}id)$
$(\!|th\text{-}state := WAITING|\!))$
$else\ th\text{-}tab)$

fun *update-th-ready*
where $update\text{-}th\text{-}ready\ th\text{-}id\ th\text{-}tab = (if\ th\text{-}id \in dom\ th\text{-}tab$
$then\ th\text{-}tab(th\text{-}id \mapsto ((the\ o\ th\text{-}tab)\ th\text{-}id)$
$(\!|th\text{-}state := READY|\!))$
$else\ th\text{-}tab)$

fun *update-th-current*
where $update\text{-}th\text{-}current\ th\text{-}id\ th\text{-}tab = (if\ th\text{-}id \in dom\ th\text{-}tab$
$then\ th\text{-}tab(th\text{-}id \mapsto ((the\ o\ th\text{-}tab)\ th\text{-}id)$

$(\!|th\text{-}state := CURRENT|\!))$
else th-tab)

fun *update-th-stopped*
where *update-th-stopped th-id th-tab* = (*if th-id* $\in$ *dom th-tab*
then th-tab(*th-id* $\mapsto$ ((*the o th-tab*) *th-id*)
$(\!|th\text{-}state := STOPPED|\!))$
else th-tab)

D.4 Get thread by thread ID

— Function that find an element in the list under a given condition

primrec *find* :: (*'a* $\Rightarrow$ *bool*) $\Rightarrow$ *'a list* $\Rightarrow$ *'a option* **where**
find - [] = *None* |
find P (*x*#*xs*) = (*if P x then Some x else find P xs*)

— A thread equality procedure ... 2 threads are equal if they have the same ID

definition *thread-eq*
where *thread-eq th-id thread* = (*th-id* = *thread-id thread*)

— An interface that let us to get a thread structure using the thread ID

definition *get-thread-by-id*
where *get-thread-by-id th-id thl*= *find* (*thread-eq th-id*) *thl*

end

theory *IPC-state-model*

imports *IPC-errors-type IPC-thread-type*

begin

E HOL representation of state type model for IPC

E.1 informations on threads

record (*'thread-id*, *'error*) *th-info* =
act-info:: *'thread-id* $\rightharpoonup$ *'error*

record $state_{id}$ = ((*int*, *int*)*memory*, $thread_{id}$, ($thread_{id}$, $thread_{ipc}$) *thread-tab*,
($thread_{id}\Rightarrow$ $thread_{id}$ $\Rightarrow$ *bool*),
($thread_{id}\Rightarrow$ (*int*, *int*)*memory* $\Rightarrow$ *bool*), *errors*) *kstate* +
th-flag ::$thread_{id}$ $\rightharpoonup$ *errors*

E.2 Interface between IPC state and threads

— An interface that let us to get a thread structure using the thread ID inside a state

definition *get-thread-by-id′*
where *get-thread-by-id′ th-id σ* = (*thread-list* (*σ*::′*a state*$_{id}$-*scheme*)) *th-id*

E.3 Interface between IPC state and memory model

definition *upd-st-res-equiv*
where *upd-st-res-equiv σ msg* = (*update-th-smm-equiv* (*current-thread σ*) (*resource σ*) *msg*)

definition *upd-st-res-equiv*$_{id}$
where *upd-st-res-equiv*$_{id}$ (*σ*::*state*$_{id}$) *msg* =
update-th-smm-equiv ((*the o* (*get-thread-by-id′ o current-thread*) *σ*) *σ*) *msg*
((*the o* (*fst o Rep-memory o resource*) *σ*) *msg*)

abbreviation
update-state caller σ f error ≡ *σ*(|*current-thread* := *caller*,
thread-list := *f caller* (*thread-list σ*),
error-codes := *error*|)

abbreviation
init-act-info caller partner σ ≡
σ(|*th-flag* := (*th-flag σ*) (*caller* := *None*, *partner*:= *None*)|)

lemma *fun-upd* (*fun-upd f x z*) *y z′* = *f*(*x*:=*z*,*y*:=*z′*)
by *auto*

lemma
assumes *1*:*x* ≠ *y*
and *2*: *fun-upd f x z* = *g*
shows *g y* = *f y*
using *assms*
by *auto*

lemma
assumes *1*:*z*≠ *None*
and *2*:*fun-upd f x z* = *g*
shows *the z* ∈ (*ran g*)
using *assms*
unfolding *ran-def*
by *auto*

end

theory *IPC-actions-preconditions*

imports *IPC-state-model*
begin

F HOL representation of IPC preconditions

F.1 IPC conditions on threads parameters

This definition assures thats the partener thread is an Open Receive thread. If this condition is not satisfied when it is checked in a given IPC stage the corresponding error code *error-IPC-21-in-PREP-SEND* is returned

definition *IPC-params-c1* ::
  $('a, 'b, \textit{th-ipc-st}, 'c, 'd, 'e)\ \textit{thread-scheme} \Rightarrow \textit{bool}$
where $\textit{IPC-params-c1 th} = (\textit{th-ipc-st th} = \textit{OR})$

lemma *IPC-params-c1-direct1*[*simp*] :
$\textit{IPC-params-c1}\ (\!|\textit{thread-id} = a_1, \textit{th-state} = a_2, \textit{th-ipc-st} = \textit{OR}, \textit{own-vmem-adr} = a_3, \textit{cpartner} = a_4|\!)$
by(*simp add*:*IPC-params-c1-def*)

lemma *IPC-params-c1-direct2*[*simp*] :
$\neg\textit{IPC-params-c1}\ (\!|\textit{thread-id} = a_1, \textit{th-state} = a_2, \textit{th-ipc-st} = \textit{CR}, \textit{own-vmem-adr} = a_3, \textit{cpartner} = a_4|\!)$
by(*simp add*:*IPC-params-c1-def*)

lemma *IPC-params-c1-direct3*[*simp*] :
$\neg\textit{IPC-params-c1}\ (\!|\textit{thread-id} = a_1, \textit{th-state} = a_2, \textit{th-ipc-st} = \textit{NR}, \textit{own-vmem-adr} = a_3, \textit{cpartner} = a_4|\!)$
by(*simp add*:*IPC-params-c1-def*)

the corresponding error code *error-IPC-22-in-PREP-SEND* is returned

definition *IPC-params-c2* ::
  $('a, 'b, \textit{th-ipc-st}, 'c, 'd, 'e)\ \textit{thread-scheme} \Rightarrow \textit{bool}$
where $\textit{IPC-params-c2 th} = (\textit{th-ipc-st th} = \textit{CR})$

the corresponding error code *error-IPC-23-in-PREP-SEND* is returned

definition *IPC-params-c3* ::
  $('a, 'b, \textit{th-ipc-st}, 'c, 'd, 'e)\ \textit{thread-scheme} \Rightarrow \textit{bool}$
where $\textit{IPC-params-c3 th} = (\textit{th-ipc-st th} = \textit{NR})$

definition *IPC-params-c4*
  $::\textit{thread}_{id} \Rightarrow \textit{thread}_{id} \Rightarrow \textit{bool}$
where $\textit{IPC-params-c4 caller partner} = (\textit{caller} \neq \textit{partner})$

definition *IPC-params-c6*
  $::\textit{thread}_{id} \Rightarrow (\textit{thread}_{id}, 'b, \textit{th-ipc-st}, 'c, \textit{thread}_{id}, 'e)\ \textit{thread-scheme} \Rightarrow \textit{bool}$
where $\textit{IPC-params-c6 caller partner} = (\textit{caller} = \textit{cpartner partner})$

definition *IPC-params-c5*
::$thread_{id}$ ⇒ *'a* $state_{id}$*-scheme* ⇒ *bool*
where *IPC-params-c5 caller σ* = (*caller* ∈ (*dom* (*thread-list σ*)) ∧
(*th-state o the*)((*thread-list σ*) *caller*) ≠ *STOPPED*)

F.2 IPC conditions on threads communication rights

definition *IPC-sub-sub-sp*
::$thread_{id}$ ⇒$thread_{id}$ ⇒ ($thread_{id}$⇒$thread_{id}$⇒ *bool*)⇒($thread_{id}$ ⇀ $thread_{ipc}$)⇒ *bool*
where *IPC-sub-sub-sp caller partner rel thl* = (*reflp rel* ∧ *rel caller partner* ∧
caller ∈ *dom thl* ∧ *partner* ∈ *dom thl*)

definition *IPC-send-comm-check*
::$thread_{id}$ ⇒$thread_{id}$ ⇒ ($thread_{id}$⇒$thread_{id}$⇒ *bool*)⇒($thread_{id}$ ⇀ $thread_{ipc}$)⇒ *bool*
where *IPC-send-comm-check caller partner rel thl* =
(*IPC-sub-sub-sp caller partner rel thl* ∧ *IPC-params-c4 caller partner*)

definition *IPC-recv-comm-check*
::$thread_{id}$ ⇒$thread_{id}$ ⇒ ($thread_{id}$⇒$thread_{id}$⇒ *bool*)⇒($thread_{id}$ ⇀ $thread_{ipc}$)⇒ *bool*
where *IPC-recv-comm-check caller partner rel thl* = *IPC-sub-sub-sp caller partner rel thl*

F.3 IPC conditions on threads access rights

definition *IPC-sub-obj-sp*
where *IPC-sub-obj-sp* = *undefined*

definition *IPC-buf-check*
:: $thread_{id}$ ⇒$thread_{id}$ ⇒ (*int*, *int*) *memory* ⇒ ($thread_{id}$ ⇒ (*int*, *int*) *memory* ⇒*bool*)⇒
($thread_{id}$ ⇀ $thread_{ipc}$) ⇒*bool*
where *IPC-buf-check caller partner mem rel thl* =
(*caller* ∈ *dom thl* ∧ *partner* ∈ *dom thl* ∧
(*dom o fst o Rep-memory*)((*own-vmem-adr o the o thl*) *caller*) ⊆
((*dom o fst o Rep-memory*) *mem*) ∧ *rel partner mem*)

definition *IPC-map-check*
where *IPC-map-check* = *undefined*

F.4 interface between IPC Preconditions and IPC *'a* $state_{id}$*-scheme*

definition *IPC-send-comm-check-*st_{id}
::$thread_{id}$ ⇒ $thread_{id}$ ⇒ *'a* $state_{id}$*-scheme*⇒ *bool*
where *IPC-send-comm-check-*st_{id} *caller partner σ* =

(*IPC-sub-sub-sp caller partner* (*communication-rights* σ) (*thread-list* σ)
$\wedge$
IPC-params-c4 caller partner)

definition *IPC-recv-comm-check-st*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *'a state*$_{id}$*-scheme*$\Rightarrow$ *bool*
where *IPC-recv-comm-check-st*$_{id}$ *caller partner* σ =
IPC-sub-sub-sp caller partner (*communication-rights* σ) (*thread-list* σ)

definition *IPC-buf-check-st*$_{id}$
::*thread*$_{id}$ $\Rightarrow$ *thread*$_{id}$ $\Rightarrow$ *'a state*$_{id}$*-scheme*$\Rightarrow$ *bool*
where *IPC-buf-check-st*$_{id}$ *caller partner* σ =
IPC-buf-check caller partner (*resource* σ) (*access-rights* σ) (*thread-list* σ)

definition *IPC-map-check-st*$_{id}$
where *IPC-map-check-st*$_{id}$ = *undefined*

end

theory *IPC-atomic-actions*
imports *IPC-actions-preconditions ../../../../src/TestLib*

begin

G HOL representation of PikeOS IPC atomic actions

G.1 Types instantiation

In order to model PikeOS IPC API atomic actions, we will instantiate types of the parameters of *a* by other Isabelle datatypes as following:

datatype *p4-stage*$_{ipc}$ =
PREP — checking file descriptor informations
| *WAIT* — synchronising
| *BUF* — MEM COPY
| *MAP* — MEM MAP
| *DONE* — IPC end

datatype (*'thread-id* , *'adresses*)
p4-direct$_{ipc}$ =
SEND 'thread-id 'thread-id 'adresses
| *RECV 'thread-id 'thread-id 'adresses*

datatype (*'thread-id* , *'adresses*) *action*$_{ipc}$*-simplified* =
IPC-SEND 'thread-id 'thread-id 'adresses

| *IPC-RECV 'thread-id 'thread-id 'adresses*

To avoid the complexe representation of memory, we represent the memory content as a list of integers and the adresses are natural numbers. An id of the thread is represented by a tuple of natural numbers that specify, the task and the partition that the thread belongs to. To use this abstraction on PikeOS IPC API in our nvironment, we will just define a new type and instantiate our free variables *a* and *b* by Isabelle natural numbers type as follwing:

type-synonym $p4\text{-}action_{ipc}\text{-}simplified = (nat \times nat \times nat\ ,\ nat\ list)\ action_{ipc}\text{-}simplified$

type-synonym $ACTION_{ipc} = (p4\text{-}stage_{ipc},\ (thread_{id},\ int\ list)\ p4\text{-}direct_{ipc})\ action_{ipc}$

type-synonym $('o,'\sigma)Mon_{SE} = '\sigma \rightharpoonup ('o * '\sigma)$

G.2 Atomic actions semantics

Actually, PikeOS IPC API provides 7 system calls. An execution of each system call will split it to atomic actions. Those atomic actions are called *stages*. In order to execute The *p4_ipc_send* call, the kernel will split it into 4 stages:

1. *PREP* stage
2. *WAIT* stage
3. *BUF* stage
4. *DONE* stage

In addition of providing interruption points, the execution of those stages is used to provide a security model to the IPC mechanism. In each stage and during the execution a set of conditions will be checked by the kernel. If one of the conditions is not satisfied, for example the communication security policy is not respected, the kernel abort the call and return an error code.

G.3 Semantics of atomic actions with thread IDs as arguments

lemma *is-part-addr-th-mem a b c= is-part-mem-th a b c*
unfolding *is-part-addr-th-mem-def is-part-mem-th-def is-part-addr-addr-def*
by (*simp*)

definition $PREP\text{-}SEND_{id}$
$::'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow 'a\ state_{id}\text{-}scheme$
where $PREP\text{-}SEND_{id}\ \sigma\ act =$

```
(case act of (IPC PREP (SEND caller partner msg)) ⇒
  if list-all ((is-part-mem-th o the) ((thread-list σ) caller) (resource σ))msg
   then
       if IPC-params-c1 ((the o thread-list σ)  partner)
       then σ(|current-thread := caller,
              thread-list    := update-th-ready caller (thread-list σ),
              error-codes    := NO-ERRORS|)
       else
        (if IPC-params-c2 ((the o thread-list σ)  partner)
         then
           if IPC-params-c6 caller ((the o thread-list σ) partner)
           then σ(|current-thread := caller,
                   thread-list    := update-th-ready caller (thread-list σ),
                  error-codes    := NO-ERRORS|)
           else
             σ(|current-thread := caller,
               thread-list    := update-th-current caller (thread-list σ),
               error-codes    := ERROR-IPC error-IPC-22-in-PREP-SEND|)
         else σ(|current-thread := caller,
               thread-list    := update-th-current caller (thread-list σ),
               error-codes    := ERROR-IPC error-IPC-23-in-PREP-SEND|))

  else σ(|current-thread := caller,
         thread-list    := update-th-current caller (thread-list σ),
    error-codes   := ERROR-MEM not-valid-sender-addr-in-PREP-SEND|))

definition PREP-RECV_id
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     PREP-RECV_id σ act = (
     case act of (IPC PREP (RECV caller partner msg)) ⇒
     if list-all ((is-part-mem-th o the) ((thread-list σ) caller) (resource σ))msg

       then
           if IPC-params-c1 ((the o thread-list σ)  partner)
           then σ(|current-thread := caller,
                  thread-list    := update-th-ready caller (thread-list σ),
                  error-codes    := NO-ERRORS|)
           else
            (if IPC-params-c2 ((the o thread-list σ)  partner)
             then
               if IPC-params-c6 caller ((the o thread-list σ) partner)
               then σ(|current-thread := caller,
                       thread-list    := update-th-ready caller (thread-list σ),
                      error-codes    := NO-ERRORS|)
               else
                 σ(|current-thread := caller ,
                   thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-IPC error-IPC-22-in-PREP-RECV|)
             else σ(|current-thread := caller ,
```

```
                    thread-list    := update-th-current caller (thread-list σ),
                   error-codes    := ERROR-IPC error-IPC-23-in-PREP-RECV⦈))

          else σ(⦇current-thread := caller ,
                thread-list    := update-th-current caller (thread-list σ),
           error-codes   := ERROR-MEM not-valid-receiver-addr-in-PREP-RECV⦈))



definition WAIT-SEND_id
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     WAIT-SEND_id σ act =
        (case act of  (IPC WAIT (SEND caller partner msg) ) ⇒
          if ¬ IPC-send-comm-check-st_id caller partner σ
          then σ (⦇current-thread := caller,
                  thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-IPC error-IPC-1-in-WAIT-SEND⦈)
          else
            if ¬ IPC-params-c4  caller  partner
            then σ (⦇current-thread := caller,
                    thread-list    := update-th-current caller (thread-list σ),
                    error-codes    := ERROR-IPC error-IPC-3-in-WAIT-SEND⦈)
            else
              if ¬IPC-params-c5 partner σ
              then
               (case (thread-list σ) caller of None ⇒
                  σ (⦇current-thread := caller,
                     thread-list    := update-th-current caller (thread-list σ),
                     error-codes    := ERROR-IPC error-IPC-6-in-WAIT-SEND⦈
                | Some th ⇒ σ(⦇current-thread := caller ,
                              thread-list    := update-th-current caller (thread-list σ),
                           error-codes   := ERROR-IPC error-IPC-5-in-WAIT-SEND⦈))

              else
                σ(⦇current-thread := caller ,
                  thread-list    := update-th-waiting caller (thread-list σ),
                  error-codes    := NO-ERRORS⦈))

definition WAIT-RECV_id
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     WAIT-RECV_id σ act =
        (case act of  (IPC WAIT (RECV caller partner msg) ) ⇒
          if ¬ IPC-recv-comm-check-st_id caller partner σ
          then σ(⦇current-thread := caller ,
                  thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-IPC error-IPC-1-in-WAIT-RECV⦈)
          else
            if ¬ IPC-params-c4  caller partner
            then σ(⦇current-thread := caller ,
```

```
                thread-list    := update-th-current caller (thread-list σ),
                error-codes    := ERROR-IPC error-IPC-3-in-WAIT-RECV⦈
          else
           if ¬ IPC-params-c5 partner σ
           then
            (case (thread-list σ) caller of None ⇒
              σ⦇current-thread := caller ,
                thread-list    := update-th-current caller (thread-list σ),
                error-codes    := ERROR-IPC error-IPC-6-in-WAIT-RECV⦈
             | Some th ⇒ σ⦇current-thread := caller ,
                          thread-list    := update-th-current caller (thread-list σ),
                          error-codes    := ERROR-IPC error-IPC-5-in-WAIT-RECV⦈)

           else
             σ⦇current-thread := caller ,
               thread-list    := update-th-waiting caller (thread-list σ),
               error-codes    := NO-ERRORS⦈)
```

abbreviation
get-th-addrs th σ ≡ (∗*thread adresses to be updated*∗)
((*sorted-list-of-set.F o Domain*) ((*own-vmem-adr o the o thread-list σ*) *th*))

abbreviation
get-msg-values msg σ ≡ (∗*the value of the addresses in a message*∗)
(*map* ((*the o* (*fst o Rep-memory*) (*resource σ*))) *msg*)

definition $BUF\text{-}SEND_{id}$
::*'a* $state_{id}$*-scheme* ⇒ $ACTION_{ipc}$ ⇒ *'a* $state_{id}$*-scheme*
where $BUF\text{-}SEND_{id}$ *σ act* =
(*case act of* (*IPC BUF* (*SEND caller partner msg*)) ⇒
if ¬ $IPC\text{-}buf\text{-}check\text{-}st_{id}$ *caller partner σ*
then *σ*⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*⦈
else
σ⦇*current-thread* := *caller*,
resource :=
foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource σ*)
(*zip* (*get-th-addrs partner σ*) (*get-msg-values msg σ*)),
thread-list := *update-th-ready caller* (*update-th-ready partner* (*thread-list σ*)),
error-codes := *NO-ERRORS*⦈
(∗*if a BUF op is execute this means that there are no errors in check stages*∗))

definition $BUF\text{-}RECV_{id}$

```
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     BUF-RECV_id σ act =
         (case act of (IPC BUF (RECV caller partner msg) ) ⇒
          if  ¬ IPC-buf-check-st_id caller partner σ
          then  σ(|current-thread := caller ,
                   thread-list    := update-th-current caller (thread-list σ),
                   error-codes    := ERROR-IPC error-IPC-1-in-BUF-RECV|)
          else
           σ(|current-thread := caller,
              resource      :=
              foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
                    (zip (get-th-addrs caller σ) (get-msg-values msg σ)),
              thread-list   := update-th-ready caller
                               (update-th-ready partner
                               (thread-list σ)),
              error-codes   := NO-ERRORS|))



definition MAP-SEND_id
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     MAP-SEND_id σ act =
         (case act of (IPC MAP (SEND caller partner msg) ) ⇒
          σ(|current-thread := caller,
             resource     :=foldl (λm (src,dst). (m (src ⋈ dst))) (resource σ)
                                  (zip msg (get-th-addrs partner σ)),
             thread-list  := update-th-ready caller
                             (update-th-ready partner
                             (thread-list σ)),
              error-codes := NO-ERRORS|)
              (*if a MAP op is execute this means that BUF was executed without
errors*))

definition MAP-RECV_id
      ::'a state_id-scheme ⇒ ACTION_ipc ⇒ 'a state_id-scheme
where     MAP-RECV_id σ act =
         (case act of (IPC MAP (RECV caller partner msg) ) ⇒
          σ(|current-thread := caller,
             resource     := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                                  (zip msg (get-th-addrs caller σ)),
             thread-list  := update-th-ready caller
                             (update-th-ready partner
                             (thread-list σ)),
              error-codes := NO-ERRORS|)
              (*if a MAP op is execute this means that BUF was executed without
errors*))
```

definition $DONE\text{-}SEND_{id}$
$::'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow 'a\ state_{id}\text{-}scheme$
where $DONE\text{-}SEND_{id}\ \sigma\ act = \sigma$

definition $DONE\text{-}RECV_{id}$
$::'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow 'a\ state_{id}\text{-}scheme$
where $DONE\text{-}RECV_{id}\ \sigma\ act = \sigma$

G.4 Semantics of atomic actions based on monads

fun $PREP\text{-}SEND_{MON} :: ACTION_{ipc} \Rightarrow 'a\ state_{id}\text{-}scheme \Rightarrow (errors * 'a\ state_{id}\text{-}scheme)\ option$
where
$PREP\text{-}SEND_{MON}\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
(*if list-all* ((*is-part-addr-th-mem o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*

then
if list-all ((*is-part-mem-th o the*) ((*thread-list* σ) *partner*) (*resource* σ))*msg*
then
if IPC-params-c1 ((*the o thread-list* σ) *partner*)
then $unit_{SE}$ (*NO-ERRORS*)
(σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
else
(*if IPC-params-c2* ((*the o thread-list* σ) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list* σ) *partner*)
then $unit_{SE}$ (*NO-ERRORS*)
(σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
else
$unit_{SE}$ (*ERROR-IPC error-IPC-22-in-PREP-SEND*)
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|))
else $unit_{SE}$ (*ERROR-IPC error-IPC-23-in-PREP-SEND*)
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|)))

else $unit_{SE}$ (*ERROR-MEM not-valid-receiver-addr-in-PREP-SEND*)
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-SEND*|))
else $unit_{SE}$ (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*)
(σ(|*current-thread* := *caller*,

```
                thread-list    := update-th-current caller (thread-list σ),
              error-codes   := ERROR-MEM not-valid-sender-addr-in-PREP-SEND|)))

  (*hyÃĺpothese: all other atomic actions have no purge*)
| PREP-SEND_MON    a σ =  unit_SE (error-codes σ ) σ


fun PREP-RECV_MON :: ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where
  PREP-RECV_MON   (IPC PREP (RECV caller partner msg)) σ =
   (if list-all ((is-part-addr-th-mem o the) ((thread-list σ) caller) (resource σ))msg

    then
      if list-all ((is-part-mem-th o the) ((thread-list σ) partner) (resource σ))msg
      then
        if IPC-params-c1 ((the o thread-list σ)  partner)
        then unit_SE (NO-ERRORS)
                  (σ(|current-thread := caller,
                     thread-list    := update-th-ready caller (thread-list σ),
                     error-codes    := NO-ERRORS|))
        else
          (if IPC-params-c2 ((the o thread-list σ)  partner)
                then
                  if IPC-params-c6 caller ((the o thread-list σ) partner)
                  then unit_SE (NO-ERRORS)
                            (σ(|current-thread := caller,
                               thread-list    := update-th-ready caller (thread-list σ),
                           error-codes    := NO-ERRORS|))
                  else
                    unit_SE (ERROR-IPC error-IPC-22-in-PREP-RECV)
                          (σ(|current-thread := caller ,
                             thread-list    := update-th-current caller (thread-list σ),
                       error-codes    := ERROR-IPC error-IPC-22-in-PREP-RECV|))
                else
                  unit_SE (ERROR-IPC error-IPC-23-in-PREP-RECV)
                    (σ(|current-thread := caller ,
                       thread-list    := update-th-current caller (thread-list σ),
                     error-codes    := ERROR-IPC error-IPC-23-in-PREP-RECV|)))

            else
              unit_SE (ERROR-MEM not-valid-receiver-addr-in-PREP-RECV)
                    (σ(|current-thread := caller ,
                       thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-MEM not-valid-receiver-addr-in-PREP-RECV|))
          else
            unit_SE (ERROR-MEM not-valid-sender-addr-in-PREP-RECV)
                  (σ(|current-thread := caller ,
                     thread-list    := update-th-current caller (thread-list σ),
```

```
                error-codes   := ERROR-MEM not-valid-sender-addr-in-PREP-RECV⦈)))

   (*hyÃÍpothese: all other atomic actions have no purge*)

| PREP-RECV_MON   a σ = unit_SE (error-codes σ ) σ

fun WAIT-SEND_MON :: ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where
  WAIT-SEND_MON  (IPC WAIT (SEND caller partner msg)) σ =
   (if ¬ IPC-send-comm-check-st_id caller partner σ
    then unit_SE (ERROR-IPC error-IPC-1-in-WAIT-SEND)
            (σ⦇current-thread := caller,
               thread-list    := update-th-current caller (thread-list σ),
               error-codes    := ERROR-IPC error-IPC-1-in-WAIT-SEND⦈)
    else
     if ¬ IPC-params-c4  caller  partner
     then  unit_SE (ERROR-IPC error-IPC-3-in-WAIT-SEND)
             (σ ⦇current-thread := caller,
                thread-list    := update-th-current caller (thread-list σ),
                error-codes    := ERROR-IPC error-IPC-3-in-WAIT-SEND⦈)
     else
      if ¬IPC-params-c5 partner σ
      then
       (case (thread-list σ) caller of None ⇒
         unit_SE (ERROR-IPC error-IPC-6-in-WAIT-SEND)
          (σ ⦇current-thread := caller,
             thread-list    := update-th-waiting caller (thread-list σ),
             error-codes    := ERROR-IPC error-IPC-6-in-WAIT-SEND⦈)
        | Some th ⇒  unit_SE (ERROR-IPC error-IPC-5-in-WAIT-SEND)
                      (σ⦇current-thread := caller ,
                         thread-list    := update-th-current caller (thread-list σ),
                   error-codes   := ERROR-IPC error-IPC-5-in-WAIT-SEND⦈))

      else
        unit_SE (NO-ERRORS) (σ⦇current-thread := caller ,
                          thread-list    := update-th-waiting caller (thread-list σ),
                           error-codes    := NO-ERRORS⦈))
| WAIT-SEND_MON   a σ =   unit_SE (error-codes σ ) σ

fun WAIT-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where WAIT-RECV_MON  (IPC WAIT (RECV caller partner msg)) σ =
      ( if ¬ IPC-recv-comm-check-st_id caller partner σ
        then unit_SE (ERROR-IPC error-IPC-1-in-WAIT-RECV)
                (σ⦇current-thread := caller ,
                   thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-IPC error-IPC-1-in-WAIT-RECV⦈)
        else
```

```
        if ¬ IPC-params-c4  caller partner
        then  unit_SE (ERROR-IPC error-IPC-3-in-WAIT-RECV)
                 (σ(|current-thread := caller ,
                    thread-list    := update-th-current caller (thread-list σ),
                    error-codes    := ERROR-IPC error-IPC-3-in-WAIT-RECV|))

        else
          if ¬ IPC-params-c5 partner σ
          then
           (case (thread-list σ) caller of None ⇒
              unit_SE (ERROR-IPC error-IPC-6-in-WAIT-RECV)
                 (σ(|current-thread := caller ,
                    thread-list    := update-th-current caller (thread-list σ),
                    error-codes    := ERROR-IPC error-IPC-6-in-WAIT-RECV|))
            | Some th ⇒ unit_SE (ERROR-IPC error-IPC-5-in-WAIT-RECV)
                          (σ(|current-thread := caller ,
                             thread-list    := update-th-current caller (thread-list
σ),
                             error-codes    := ERROR-IPC error-IPC-5-in-WAIT-RECV|)))

          else
            unit_SE (NO-ERRORS)
               (σ(|current-thread := caller ,
                  thread-list    := update-th-waiting caller (thread-list σ),
                  error-codes    := NO-ERRORS|)))

| WAIT-RECV_MON  a σ = unit_SE (error-codes σ ) σ

fun BUF-SEND_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where
  BUF-SEND_MON  (IPC BUF (SEND caller partner msg)) σ =
    (if  ¬ IPC-buf-check-st_id caller partner σ
     then  unit_SE (ERROR-IPC error-IPC-1-in-BUF-RECV)
              (σ(|current-thread := caller ,
                  thread-list    := update-th-current caller (thread-list σ),
                  error-codes    := ERROR-IPC error-IPC-1-in-BUF-RECV|))
     else
       unit_SE (NO-ERRORS)
            (σ(|current-thread := caller,
              resource      := update-list (resource σ)
                                    (zip  ((sorted-list-of-set.F o dom  o fst o
Rep-memory)
                                ((own-vmem-adr o the o thread-list σ) partner))
                                     (map ((the o (fst o Rep-memory) (resource
σ))) msg)),
              thread-list   := update-th-ready caller
                               (update-th-ready partner
                               (thread-list σ)),
```

```
            error-codes   := NO-ERRORS⦈)))
| BUF-SEND_MON   a σ =  unit_SE (error-codes σ ) σ

fun BUF-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where
  BUF-RECV_MON  (IPC BUF (RECV caller partner msg)) σ =
    ( if  ¬ IPC-buf-check-st_id caller partner σ
      then
        unit_SE (ERROR-IPC error-IPC-1-in-BUF-RECV)
        (σ⦇current-thread := caller ,
           thread-list    := update-th-current caller (thread-list σ),
           error-codes    := ERROR-IPC error-IPC-1-in-BUF-RECV⦈)
      else
       unit_SE (NO-ERRORS)
       (σ⦇current-thread := caller,
          resource       := update-list (resource σ)
                                        (zip  ((sorted-list-of-set.F o dom  o fst o
Rep-memory)
                                    ((own-vmem-adr o the o thread-list σ) caller))
                                    (map ((the o (fst o Rep-memory) (resource σ)))
msg)),
          thread-list   := update-th-ready caller
                           (update-th-ready partner
                           (thread-list σ)),
          error-codes   := NO-ERRORS⦈)))
| BUF-RECV_MON  a σ =  unit_SE (error-codes σ ) σ


fun MAP-SEND_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where MAP-SEND_MON    (IPC MAP (SEND caller partner msg) ) σ =
      unit_SE (NO-ERRORS ) (σ⦇current-thread := caller,
                            resource     := foldl (λm (src,dst). (m (src ⋈ dst)))
(resource σ)
                             (zip msg (get-th-addrs partner σ)),
                           thread-list  := update-th-ready caller
                                           (update-th-ready partner
                                           (thread-list σ)),
                            error-codes := NO-ERRORS⦈)
    | MAP-SEND_MON  a σ =  unit_SE (error-codes σ ) σ


fun MAP-RECV_MON ::ACTION_ipc ⇒ 'a state_id-scheme ⇒ (errors * 'a state_id-scheme)
option
where MAP-RECV_MON    (IPC MAP (SEND caller partner msg) ) σ =
      unit_SE (NO-ERRORS)
            (σ⦇current-thread := caller,
               resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
```

$(zip\ msg\ (get\text{-}th\text{-}addrs\ caller\ \sigma)),$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ := NO\text{-}ERRORS\|)$
$|\ MAP\text{-}RECV_{MON}\ \ a\ \sigma =\ \ unit_{SE}\ (error\text{-}codes\ \sigma\ )\ \sigma$

fun $DONE\text{-}SEND_{MON}$ $::ACTION_{ipc} \Rightarrow {}'a\ state_{id}\text{-}scheme \Rightarrow (errors * {}'a\ state_{id}\text{-}scheme)$ *option*
where $DONE\text{-}SEND_{MON}\ \ a\ \sigma = unit_{SE}\ (error\text{-}codes\ \sigma\ )\ \sigma$

fun $DONE\text{-}RECV_{MON}$ $::ACTION_{ipc} \Rightarrow {}'a\ state_{id}\text{-}scheme \Rightarrow (errors * {}'a\ state_{id}\text{-}scheme)$ *option*
where $DONE\text{-}RECV_{MON}\ \ a\ \sigma = unit_{SE}\ (error\text{-}codes\ \sigma\ )\ \sigma$

definition $IPC\text{-}protocol\ a =$
$(out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ (out2 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ (out3 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;$
$(out4 \leftarrow WAIT\text{-}RECV_{MON}\ a; (out5 \leftarrow BUF\text{-}SEND_{MON}\ a; (out6 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;$
$(out7 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ \ DONE\text{-}RECV_{MON}\ a)))))))$

G.5 Execution function for PikeOS IPC atomic actions with thread IDs as arguments

fun $exec\text{-}action_{id}$
$::{}'a\ state_{id}\text{-}scheme \Rightarrow ACTION_{ipc} \Rightarrow {}'a\ state_{id}\text{-}scheme$
where
$PREP\text{-}SEND\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg)) =$
$PREP\text{-}SEND_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))|$

$PREP\text{-}RECV\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))|$

$WAIT\text{-}SEND\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ WAIT(SEND\ caller\ partner\ msg))\ =$
$WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))|$

$WAIT\text{-}RECV\text{-}run'{:}exec\text{-}action_{id}\ \sigma\ (IPC\ WAIT(RECV\ caller\ partner\ msg))\ =$
$WAIT\text{-}RECV_{id}\ \sigma\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))|$

$BUF\text{-}SEND\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ =$
$BUF\text{-}SEND_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ |$

$BUF\text{-}RECV\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ BUF(RECV\ caller\ partner\ msg))\ \ =$
$BUF\text{-}RECV_{id}\ \sigma\ (IPC\ BUF(RECV\ caller\ partner\ msg))\ \ |$
$MAP\text{-}SEND\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ =$
$MAP\text{-}SEND_{id}\ \sigma\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ |$

$MAP\text{-}RECV\text{-}run'\ {:}exec\text{-}action_{id}\ \sigma\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \ =$

$$MAP\text{-}RECV_{id}\ \sigma\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))$$

|

$$DONE\text{-}SEND\text{-}run'\ :exec\text{-}action_{id}\ \sigma\ (IPC\ DONE(SEND\ caller\ partner\ msg)) = \sigma$$

|

$$DONE\text{-}RECV\text{-}run'\ :exec\text{-}action_{id}\ \sigma\ (IPC\ DONE(RECV\ caller\ partner\ msg)) = \sigma$$

G.6 Predicates on atomic actions

Different cases of send action

definition *actions-send-cases a caller partner msg* =($a = IPC\ PREP\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ (SEND\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ (SEND\ caller\ partner\ msg)$)

Different cases of receive action

definition *actions-receiv-cases a caller partner msg* =($a = IPC\ PREP\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ WAIT\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ BUF\ (RECV\ caller\ partner\ msg) \vee$
$a = IPC\ DONE\ (RECV\ caller\ partner\ msg)$)

A comparison procedure between actions. Used to indentify actions that can reply to an aborted system call.

definition *actioneq-op a a'* = (*case a of*
$(IPC\ PREP\ (SEND\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}receiv\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ PREP\ (RECV\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}send\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ WAIT\ (SEND\ caller\ partner\ msg)\ ) \Rightarrow$
$(actions\text{-}receiv\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ WAIT\ (RECV\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}send\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ BUF\ (SEND\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}receiv\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ BUF\ (RECV\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}send\text{-}cases\ a'\ partner\ caller\ msg)$
| $(IPC\ DONE\ (SEND\ caller\ partner\ msg)) \Rightarrow$
$(actions\text{-}receiv\text{-}cases\ a'\ partner\ caller\ msg)$

$$
\begin{array}{l}
\quad \mid (\mathit{IPC\ DONE}\ (\mathit{RECV\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}send\text{-}cases}\ a'\ \mathit{partner\ caller\ msg}) \\
\\
)
\end{array}
$$

A comparison procedure between actions. Used to indentify actions that will be aborted.

definition $\mathit{actioneq}\ a\ a' = (\mathit{case}\ a\ \mathit{of}$

$$
\begin{array}{l}
\quad (\mathit{IPC\ PREP}\ (\mathit{SEND\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}send\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ PREP}\ (\mathit{RECV\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}receiv\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ WAIT}\ (\mathit{SEND\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}send\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ WAIT}\ (\mathit{RECV\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}receiv\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ BUF}\ (\mathit{SEND\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}send\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ BUF}\ (\mathit{RECV\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}receiv\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ DONE}\ (\mathit{SEND\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}send\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
\mid (\mathit{IPC\ DONE}\ (\mathit{RECV\ caller\ partner\ msg})) \Rightarrow \\
\qquad (\mathit{actions\text{-}receiv\text{-}cases}\ a'\ \mathit{caller\ partner\ msg}) \\
)
\end{array}
$$

G.7 Lemmas and simplification rules related to atomic actions

lemma *mem-inv1*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) = *resource* σ
apply (*auto simp* : *WAIT-SEND*$_{id}$*-def*)
apply (*cases thread-list* σ *caller*,*auto*)
done

lemma *mem-inv2*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) = *resource* σ
apply (*auto simp* : *WAIT-RECV*$_{id}$*-def*)
apply (*cases thread-list* σ *caller*,*auto*)
done

lemma *mem-inv3*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) = *resource* σ
by (*auto simp* : *PREP-RECV*$_{id}$*-def*)

lemma *mem-inv4*[*simp*]:

resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) = *resource* σ
by (*auto simp* : *PREP-SEND*$_{id}$*-def*)

lemma *mem-inv5*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC BUF*(*RECV caller partner msg*))) =
(*if* ¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ
then resource σ
else foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)))
by (*auto simp* : *BUF-RECV*$_{id}$*-def*)

lemma *mem-inv5-E*:
assumes *1*: σ' = *resource* (*exec-action*$_{id}$ σ (*IPC BUF*(*RECV caller partner msg*)))
and *2*: ¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ ⟹ σ' = *resource* σ ⟹ *Q*
and *3*: *IPC-buf-check-st*$_{id}$ *caller partner* σ ⟹
σ' = *foldl* (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)) ⟹ *Q*
shows *Q*
proof −
show *?thesis*
using *1* **unfolding** *mem-inv5*
proof (*cases* ¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ)
case *True*
show *?thesis*
using *True 1* **unfolding** *mem-inv5*
by (*simp*, *elim 2*)
next
case *False*
show *?thesis*
using *False 1* **unfolding** *mem-inv5*
by (*simp*, *elim 3*, *simp*)
qed
qed

lemma *mem-inv6*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC BUF*(*SEND caller partner msg*))) =
(*if* ¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ
then resource σ
else foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs partner* σ) (*get-msg-values msg* σ)))
by (*auto simp* :*BUF-SEND*$_{id}$*-def*)

lemma *mem-inv6-E*:
assumes *1*: σ' = *resource* (*exec-action*$_{id}$ σ (*IPC BUF*(*SEND caller partner msg*)))
and *2*: ¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ ⟹ σ' = *resource* σ ⟹ *Q*
and *3*: *IPC-buf-check-st*$_{id}$ *caller partner* σ ⟹

```
        σ' = foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
              (zip (get-th-addrs partner σ) (get-msg-values msg σ)) ⟹ Q
  shows  Q
proof −
  show ?thesis
  using 1 unfolding mem-inv5
    proof (cases ¬ IPC-buf-check-st_id caller partner σ )
      case True
      show ?thesis
      using True 1 unfolding mem-inv6
      by (simp, elim 2)
    next
      case False
      show ?thesis
      using False 1 unfolding mem-inv6
      by (simp, elim 3, simp)
   qed
qed

lemma mem-inv7[simp]:
  resource (exec-action_id σ (IPC DONE(SEND caller partener msg))) = resource
σ
  by simp

lemma mem-inv8[simp]:
  resource (exec-action_id σ (IPC DONE(RECV caller partener msg))) = resource
σ
  by simp

lemma mem-inv9[simp]:
     resource (exec-action_id σ (IPC PREP(SEND caller partener msg))) =
      resource (exec-action_id σ (IPC PREP(RECV caller partener msg)))
  unfolding mem-inv3 mem-inv4
  by simp

lemma mem-inv10[simp]:
     resource (exec-action_id σ (IPC PREP(SEND caller partener msg))) =
      resource (exec-action_id σ (IPC WAIT(SEND caller partener msg)))
  unfolding mem-inv4 mem-inv1
  by simp

lemma mem-inv11[simp]:
     resource (exec-action_id σ (IPC PREP(SEND caller partener msg))) =
      resource (exec-action_id σ (IPC WAIT(RECV caller partener msg)))
  unfolding mem-inv2 mem-inv4
  by simp

lemma mem-inv12[simp]:
     resource (exec-action_id σ (IPC PREP(SEND caller partener msg))) =
```

resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv4*
by *simp*

lemma *mem-inv13*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv4*
by *simp*

lemma *mem-inv14*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*)))
unfolding *mem-inv3 mem-inv1*
by *simp*

lemma *mem-inv15*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*)))
unfolding *mem-inv2 mem-inv3*
by *simp*

lemma *mem-inv16*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC PREP*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv3*
by *simp*

lemma *mem-inv17*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv1*
by *simp*

lemma *mem-inv18*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv1*
by *simp*

lemma *mem-inv19*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*)))
unfolding *mem-inv2*
by *simp*

lemma *mem-inv20*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC WAIT*(*RECV caller partener msg*))) =

resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
unfolding *mem-inv2*
by *simp*

lemma *mem-inv21*[*simp*]:
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*SEND caller partener msg*))) =
resource (*exec-action*$_{id}$ σ (*IPC DONE*(*RECV caller partener msg*)))
by *simp*

G.8 Composition equality on same action

For the general case the order of the executions of PikeOS matter iff executed on the same action, because the semantics of the execution related to each action is separated

lemma *sem-comp-prep-send1*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*) = (*out1* ← *PREP-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send2*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*) = (*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send3*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*) = (*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send4*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send5*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send6*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send7*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send8*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send9*:
(*out1* ← *PREP-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv2*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*) = (*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv3*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*) = (*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv4*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*, *simp-all add*: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv5*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv6*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv7*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv8*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv9*:
(*out1* ← *PREP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send4*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send5*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send6*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send7*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ;*MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send8*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send9*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a*; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv4*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv5*:

$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ BUF\text{-}RECV_{MON}\ a) = (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv6*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}SEND_{MON}\ a) = (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv7*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ MAP\text{-}RECV_{MON}\ a) = (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv8*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv9*:
$(out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ DONE\text{-}RECV_{MON}\ a) = (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ WAIT\text{-}RECV_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send6*:
$(out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ DONE\text{-}SEND_{MON}\ a) = (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ BUF\text{-}SEND_{MON}\ a)$
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send7*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a*; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send8*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send9*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv6*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv7*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv8*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*) = (*out1* ← *MAP-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$*.induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv9*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*) = (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$.*induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send6*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send7*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*) = (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send8*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*) = (*out1* ← *BUF-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$.*induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send9*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*) = (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$.*induct*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$.*induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*, *rule p4-direct*$_{ipc}$.*induct*,

simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv6*:
(*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*) = (*out1* ← *DONE-SEND*$_{MON}$

a ; $MAP\text{-}RECV_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv7*:
($out1 \leftarrow MAP\text{-}RECV_{MON}$ a ; $DONE\text{-}RECV_{MON}$ a) = ($out1 \leftarrow DONE\text{-}RECV_{MON}$ a ; $MAP\text{-}RECV_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

G.9 Composition equality on different same actions: partial order reduction

For the specific case of IPC protocol the order of the executions of PikeOS does matter iff executed on different actions, because the semantics of the execution related to each action can react in some cases on the same field of the state, eg: the field related to erro codes... So the switch between the execution order related to IPC actions can be done but under some assumptions and only for a subset of actions

lemma *sem-comp-prep-send10*:
($out1 \leftarrow PREP\text{-}SEND_{MON}$ a ; $DONE\text{-}SEND_{MON}$ b) = ($out1 \leftarrow DONE\text{-}SEND_{MON}$ b ; $PREP\text{-}SEND_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-send11*:
($out1 \leftarrow PREP\text{-}SEND_{MON}$ a ; $DONE\text{-}RECV_{MON}$ b) = ($out1 \leftarrow DONE\text{-}RECV_{MON}$ b ; $PREP\text{-}SEND_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv10*:
($out1 \leftarrow PREP\text{-}RECV_{MON}$ a ; $DONE\text{-}SEND_{MON}$ b) = ($out1 \leftarrow DONE\text{-}SEND_{MON}$ b ; $PREP\text{-}RECV_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-prep-recv11*:
($out1 \leftarrow PREP\text{-}RECV_{MON}$ a ; $DONE\text{-}RECV_{MON}$ b) = ($out1 \leftarrow DONE\text{-}RECV_{MON}$ b ; $PREP\text{-}RECV_{MON}$ a)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

term (*resource o snd o the*) ((*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *b*) *σ*)

lemma *WAIT-SEND*$_{MON}$*-None*: *WAIT-SEND*$_{MON}$ (*IPC WAIT a*) *σ* ≠ *None*
by (*induct a*, *auto simp add*: *unit-SE-def split*:*option.split*)

lemma *WAIT-RECV*$_{MON}$*-None*: *WAIT-RECV*$_{MON}$ (*IPC WAIT a*) *σ* ≠ *None*
by (*induct a*, *auto simp add*: *unit-SE-def split*:*option.split*)

lemma *sem-comp-wait-send10*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-send11*:
(*out1* ← *WAIT-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b*; *WAIT-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv10*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-wait-recv11*:
(*out1* ← *WAIT-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *WAIT-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send10*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-send11*:
(*out1* ← *BUF-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b*; *BUF-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv10*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-buf-recv11*:
(*out1* ← *BUF-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *BUF-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send10*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-send11*:
(*out1* ← *MAP-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *MAP-SEND*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv8*:
(*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *b*) = (*out1* ← *DONE-SEND*$_{MON}$ *b* ; *MAP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

lemma *sem-comp-map-recv9*:
(*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *b*) = (*out1* ← *DONE-RECV*$_{MON}$ *b* ; *MAP-RECV*$_{MON}$ *a*)
by (*rule ext*, *induct a*, *rule p4-stage*$_{ipc}$*.induct*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def*, *rule p4-direct*$_{ipc}$*.induct*,
simp-all add: *unit-SE-def bind-SE-def split*:*option.split*)

end

theory *IPC-traces*

imports *IPC-atomic-actions*

begin

H HOL representation of PikeOS IPC traces

type-synonym *trace*$_{ipc}$ = *ACTION*$_{ipc}$ *list*

H.1 Execution function for PikeOS IPC traces

definition *exec-action*$_{id}$*-Mon*
where *exec-action*$_{id}$*-Mon* = (λ*actl st*. *Some* (*error-codes*(*exec-action*$_{id}$ *st actl*),
exec-action$_{id}$ *st actl*))

H.2 Trace refinement

H.3 Execution function for actions with thread ID

lemma (((*th-flag σ*) *caller*) = *None*) = (*caller* ∉*dom* (*th-flag σ*))
by *auto*

lemma
assumes *caller* ∈*dom* (*th-flag σ*)
shows *the*((*th-flag σ*) *caller*) ∈ *ran* (*th-flag σ*)
using *assms*
by (*auto simp*: *ranI*)

abbreviation
get-caller-error caller σ ≡ (*the o*(*th-flag*) *σ*) *caller*

abbreviation
remove-caller-error caller σ ≡ *σ*(|*th-flag* := (*th-flag σ*) (*caller* := *None*)
|)

abbreviation

set-caller-partner-error caller partner σ σ' *out'* $\equiv$
$\sigma'(\!|$*th-flag* := (*th-flag* σ)
(*caller* := *Some*(*out'*(∗*just* (*a*,*out'*)*?*∗)),
partner:= *Some* (*out'*(∗*just* (*a*,*out'*)*?*∗)))
$|\!)$

abbreviation
error-tab-transfer caller σ σ' $\equiv$ $\sigma'(\!|$*th-flag* := (*th-flag* σ)$|\!)$

abbreviation
set-no-error-preps caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-waits caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-bufs caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-dones caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-prepr caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-waitr caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-no-error-bufr caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$*NO-ERRORS*)$|\!)$

abbreviation
set-no-error-doner caller partner σ σ' *msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ (*caller* $\mapsto$ *NO-ERRORS*)$|\!)$

abbreviation
set-error-mem-preps caller partner σ σ' *error-mem msg* $\equiv$
σ' $(\!|state_{id}$.*th-flag* := $state_{id}$.*th-flag* σ(*caller* $\mapsto$ *ERROR-MEM error-mem*,
partner $\mapsto$ *ERROR-MEM error-mem*)$|\!)$

abbreviation
set-error-mem-waits caller partner σ σ' *error-mem msg* $\equiv$

σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-bufs caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-maps caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|th\text{-}flag :=th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-dones caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-prepr caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-waitr caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-bufr caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-mapr caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-mem-doner caller partner $\sigma\ \sigma'$ *error-mem msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}MEM\ error\text{-}mem,$
$partner \mapsto ERROR\text{-}MEM\ error\text{-}mem)|\!$)

abbreviation
set-error-ipc-preps caller partner $\sigma\ \sigma'$ *error-ipc msg* $\equiv$
σ' ($\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!$)
abbreviation

set-error-ipc-waits caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$

abbreviation
set-error-ipc-bufs caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$
abbreviation
set-error-ipc-maps caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$
abbreviation
set-error-ipc-dones caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$

abbreviation
set-error-ipc-prepr caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\ (caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$
abbreviation
set-error-ipc-waitr caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$

abbreviation
set-error-ipc-bufr caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$
abbreviation
set-error-ipc-mapr caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$
abbreviation
set-error-ipc-doner caller partner σ σ' *error-ipc msg* $\equiv$
σ' $(\!|state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma(caller \mapsto ERROR\text{-}IPC\ error\text{-}ipc,$
$partner \mapsto ERROR\text{-}IPC\ error\text{-}ipc)|\!)$

fun $abort_{lift} :: (ACTION_{ipc} \Rightarrow (errors,\ 'a\ state_{id}\text{-}scheme)Mon_{SE}) \Rightarrow$
$(ACTION_{ipc} \Rightarrow (errors,\ 'a\ state_{id}\text{-}scheme)Mon_{SE})$
where $abort_{lift}$ *ioprog a* σ =
(*case a of*
(*IPC DONE* (*SEND caller partner msg*)) $\Rightarrow$
if caller $\in$ *dom* (*th-flag* σ) (∗*should add the condition*: *in which action ID*
the error occurs∗)
then Some((*the*((*th-flag* σ) *caller*))(∗*shoud be*: *my error*∗),

```
                    σ(|th-flag := (th-flag σ)(caller := None)
                                |])
            else (case ioprog a σ of
                    None ⇒ None (∗never happens in our exec fun∗)
                  | Some(out′, σ′) ⇒ Some(NO-ERRORS, σ)) (∗execute done∗)
        | (IPC DONE (RECV caller partner msg)) ⇒
            if caller ∈ dom  (th-flag σ)
            then Some((the((th-flag σ) caller))(∗shoud be: my error∗),
                    σ(|th-flag := (th-flag σ) (caller := None)|))
            else (case ioprog a σ of
                    None ⇒ None (∗never happens in our exec fun∗)
                  | Some(out′, σ′) ⇒ Some(NO-ERRORS, σ)) (∗execute done∗)
        | (IPC - (SEND caller partner msg)) ⇒
             if caller ∈ dom (th-flag σ)
             then Some(get-caller-error caller σ(∗should be: my error∗), σ) (∗ purge
∗)
             else (case ioprog a σ of
                    None ⇒ None (∗never happens in our exec fun∗)
                | Some(NO-ERRORS, σ′) ⇒ Some(NO-ERRORS, error-tab-transfer
caller σ σ′)
                  | Some(ERROR-MEM error-memory, σ′) ⇒
                     Some(ERROR-MEM error-memory,
                          set-caller-partner-error caller partner σ σ′ (ERROR-MEM
error-memory) )
                  | Some(ERROR-IPC error-IPC, σ′) ⇒
                     Some(ERROR-IPC error-IPC,
                           set-caller-partner-error caller partner σ σ′ (ERROR-IPC
error-IPC) ))
                   (∗both caller and partner were ′informed′ to be in error−state.∗)
        | (IPC - (RECV caller partner msg)) ⇒
             if caller ∈ dom (th-flag σ)
            then Some(get-caller-error caller σ (∗should be: my error∗), σ) (∗purge∗)

             else (case ioprog a σ of
                    None ⇒ None (∗never happens in our exec fun∗)
                | Some(NO-ERRORS, σ′) ⇒ Some(NO-ERRORS, error-tab-transfer
caller σ σ′)
                  | Some(ERROR-MEM error-memory, σ′) ⇒
                     Some(ERROR-MEM error-memory,
                          set-caller-partner-error caller partner σ σ′ (ERROR-MEM
error-memory) )
                  | Some(ERROR-IPC error-IPC, σ′) ⇒
                     Some(ERROR-IPC error-IPC,
                           set-caller-partner-error caller partner σ σ′ (ERROR-IPC
error-IPC) ))
                   (∗both caller and partner were ′informed′ to be in error−state.∗)

       (∗hypothese: all other atomic actions have no purge∗)
    )
```

lemma *exec-action$_{id}$-Mon-th-flag0*:
a = IPC ipc-stage (ipc-direction) ⟹ ipc-stage ≠ DONE ⟹
exec-action$_{id}$-Mon a σ = Some (NO-ERRORS,σ′) ⟹ th-flag σ = th-flag σ′
unfolding *exec-action$_{id}$-Mon-def*
apply *auto*
apply (*cases ipc-stage*)
apply (*case-tac ipc-direction*)
apply *simp-all*
unfolding *PREP-SEND$_{id}$-def PREP-RECV$_{id}$-def*
apply *simp-all*
apply (*case-tac ipc-direction*)
apply *simp-all*
unfolding *WAIT-SEND$_{id}$-def*
apply *simp-all*
apply *safe*
apply (*case-tac thread-list σ* (*a*, *aa*, *b*))
apply *simp-all*
unfolding *WAIT-RECV$_{id}$-def*
apply *simp-all*
apply *safe*
apply *simp-all*
apply (*case-tac thread-list σ* (*a*, *aa*, *b*))
apply *simp-all*
apply (*case-tac ipc-direction*)
apply *simp-all*
unfolding *BUF-SEND$_{id}$-def*
apply *simp-all*
unfolding *BUF-RECV$_{id}$-def*
apply *simp-all*
apply (*cases ipc-direction*)
apply (*simp-all add*: *MAP-SEND$_{id}$-def MAP-RECV$_{id}$-def*)
done

H.4 IPC operations with thread ID

We define an *operation* as a trace with a given order on atomic actions. For the IPC API we will define two types of operations, we call the first type *request* and the second type *reply*. Following this terminology a given PikeOS thread can request to communicate with another thread or reply to a communication request. The Isabelle specification of operations is as

following:

definition *ipc-send-request*$_{id}$
:: *thread*$_{id}$⇒ *int list*⇒ *thread*$_{id}$⇒ *trace*$_{ipc}$ ((- ▷$_{id}$ - ▷$_{id}$/ -) [*201*, *0*, *201*] *200*)
where
caller ▷$_{id}$ *msg* ▷$_{id}$ *partner* ≡ [*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*)]

definition *ipc-recv-request*$_{id}$
:: *thread*$_{id}$⇒ *int list*⇒ *thread*$_{id}$⇒ *trace*$_{ipc}$ ((- ◁$_{id}$ - ◁$_{id}$/ -) [*201*, *0*, *201*] *200*)
where
caller ◁$_{id}$ *msg* ◁$_{id}$ *partner* ≡ [*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*)]

— A thread can do response operation to sending or receiving message response

definition *ipc-send-response*$_{id}$
::*thread*$_{id}$⇒ *int list*⇒ *thread*$_{id}$⇒ *trace*$_{ipc}$ ((- ▷$_{id}$ - ▷$_{id}$/ -) [*201*, *0*, *201*] *200*)
where
caller ▷$_{id}$ *msg* ▷$_{id}$ *partner* ≡ [*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*),
IPC DONE (*RECV partner caller msg*)]

definition *ipc-recv-response*$_{id}$
::*thread*$_{id}$⇒ *int list*⇒ *thread*$_{id}$⇒ *trace*$_{ipc}$ ((- ◁$_{id}$ - ◁$_{id}$/ -) [*201*, *0*, *201*] *200*)
where
caller ◁$_{id}$ *msg* ◁$_{id}$ *partner* ≡ [*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC DONE (*SEND partner caller msg*),
IPC DONE (*RECV caller partner msg*)]

lemmas *request-normalizer* =
ipc-send-response$_{id}$*-def ipc-recv-response*$_{id}$*-def ipc-send-request*$_{id}$*-def ipc-recv-request*$_{id}$*-def*

H.5 IPC operations with free variables

abbreviation *ipc-send-request* ((- ▷ - ▷/ -) [*201*, *0*, *201*] *200*)
where *caller* ▷ *msg* ▷ *partner* ≡ [*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*)]

abbreviation *ipc-recv-request* ((- ◁ - ◁/ -) [*201*, *0*, *201*] *200*)
where *caller* ◁ *msg* ◁ *partner* ≡ [*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*)]

abbreviation *ipc-send-response* ((- $\trianglerighteq$ - $\trianglerighteq$/ -) [*201*, *0*, *201*] *200*)
where *caller* $\trianglerighteq$ *msg* $\trianglerighteq$ *partner* $\equiv$ [*IPC PREP* (*SEND caller partner msg*),
IPC WAIT (*SEND caller partner msg*),
IPC BUF (*SEND caller partner msg*),
IPC MAP (*SEND caller partner msg*),
IPC DONE (*SEND caller partner msg*),
IPC DONE (*RECV partner caller msg*)]

abbreviation *ipc-recv-response* ((- $\trianglelefteq$ - $\trianglelefteq$/ -) [*201*, *0*, *201*] *200*)
where
caller $\trianglelefteq$ *msg* $\trianglelefteq$ *partner* $\equiv$ [*IPC PREP* (*RECV caller partner msg*),
IPC WAIT (*RECV caller partner msg*),
IPC BUF (*RECV caller partner msg*),
IPC MAP (*RECV caller partner msg*),
IPC DONE (*SEND partner caller msg*),
IPC DONE (*RECV caller partner msg*)]

H.6 Pridicates on operations

definition *is-ipc-trace*
where *is-ipc-trace actl* = ($\forall$ *a*$\in$*set*(*actl*::*trace*$_{ipc}$).$\exists$ *caller partner msg*.
a = *IPC PREP* (*RECV caller partner msg*)$\vee$
a = *IPC WAIT* (*RECV caller partner msg*)$\vee$
a = *IPC BUF* (*RECV caller partner msg*)$\vee$
a = *IPC DONE* (*RECV caller partner msg*)$\vee$
a = *IPC PREP* (*SEND caller partner msg*)$\vee$
a = *IPC WAIT* (*SEND caller partner msg*)$\vee$
a = *IPC BUF* (*SEND caller partner msg*)$\vee$
a = *IPC DONE* (*SEND caller partner msg*))
definition *is-ipc-trace*$_{id}$
where *is-ipc-trace*$_{id}$ *actl* = ($\forall$ *a*$\in$*set*(*actl*::*trace*$_{ipc}$).$\exists$ *caller partner msg*.
a = *IPC PREP* (*RECV caller partner msg*)$\vee$
a = *IPC WAIT* (*RECV caller partner msg*)$\vee$
a = *IPC BUF* (*RECV caller partner msg*)$\vee$
a = *IPC DONE* (*RECV caller partner msg*)$\vee$
a = *IPC PREP* (*SEND caller partner msg*)$\vee$
a = *IPC WAIT* (*SEND caller partner msg*)$\vee$
a = *IPC BUF* (*SEND caller partner msg*)$\vee$
a = *IPC DONE* (*SEND caller partner msg*))

H.7 Simplification rules related to traces

lemma *prep-send-comp-mbind-eq2*:
mbind is (λ*a*. (*out1* $\leftarrow$ *PREP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ =
mbind is (λ*a*. (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *PREP-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-send1*)

lemma *prep-send-comp-mbind-eq3*:

mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ WAIT\text{-}SEND_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send2*)

lemma *prep-send-comp-mbind-eq4*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ WAIT\text{-}RECV_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send3*)

lemma *prep-send-comp-mbind-eq5*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ BUF\text{-}SEND_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow BUF\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send4*)

lemma *prep-send-comp-mbind-eq6*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ BUF\text{-}RECV_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow BUF\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send5*)

lemma *prep-send-comp-mbind-eq7*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ MAP\text{-}SEND_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow MAP\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send6*)

lemma *prep-send-comp-mbind-eq8*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ MAP\text{-}RECV_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow MAP\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send7*)

lemma *prep-send-comp-mbind-eq9*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ DONE\text{-}SEND_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow DONE\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send8*)

lemma *prep-send-comp-mbind-eq10*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}SEND_{MON}\ a\ ;\ \ DONE\text{-}RECV_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow DONE\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}SEND_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-send9*)

lemma *prep-recv-comp-mbind-eq1*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ \ WAIT\text{-}SEND_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow WAIT\text{-}SEND_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$
by (*simp only*: *sem-comp-prep-recv2*)

lemma *prep-recv-comp-mbind-eq2*:
mbind is $(\lambda a.\ (out1 \leftarrow PREP\text{-}RECV_{MON}\ a\ ;\ \ WAIT\text{-}RECV_{MON}\ a))\ \sigma =$
mbind is $(\lambda a.\ (out1 \leftarrow WAIT\text{-}RECV_{MON}\ a\ ;\ PREP\text{-}RECV_{MON}\ a))\ \sigma$

by (*simp only*: *sem-comp-prep-recv3*)

lemma *prep-recv-comp-mbind-eq3*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv4*)

lemma *prep-recv-comp-mbind-eq4*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv5*)

lemma *prep-recv-comp-mbind-eq5*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv6*)

lemma *prep-recv-comp-mbind-eq6*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv7*)

lemma *prep-recv-comp-mbind-eq7*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv8*)

lemma *prep-recv-comp-mbind-eq8*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *PREP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ *a* ; *PREP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-prep-recv9*)

lemma *wait-send-comp-mbind-eq1*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send4*)

lemma *wait-send-comp-mbind-eq2*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send5*)

lemma *wait-send-comp-mbind-eq3*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send6*)

lemma *wait-send-comp-mbind-eq4*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *MAP-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send7*)

lemma *wait-send-comp-mbind-eq5*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send8*)

lemma *wait-send-comp-mbind-eq6*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-SEND*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ *a* ; *WAIT-SEND*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-send9*)

lemma *wait-recv-comp-mbind-eq1*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *BUF-SEND*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv4*)

lemma *wait-recv-comp-mbind-eq2*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv5*)

lemma *wait-recv-comp-mbind-eq3*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *MAP-SEND*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv6*)

lemma *wait-recv-comp-mbind-eq4*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *MAP-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv7*)

lemma *wait-recv-comp-mbind-eq5*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv8*)

lemma *wait-recv-comp-mbind-eq6*:
mbind is (λa. (*out1* $\leftarrow$ *WAIT-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*)) σ =
mbind is (λa. (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ *a* ; *WAIT-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-wait-recv9*)

lemma *buf-send-comp-mbind-eq1*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ a ; *DONE-SEND*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ a ; *BUF-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-send6*)

lemma *buf-send-comp-mbind-eq2*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ a ; *DONE-RECV*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ a ; *BUF-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-send7*)

lemma *buf-send-comp-mbind-eq3*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ a ; *MAP-SEND*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ a ; *BUF-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-send8*)

lemma *buf-send-comp-mbind-eq4*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-SEND*$_{MON}$ a ; *MAP-RECV*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-RECV*$_{MON}$ a ; *BUF-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-send9*)

lemma *map-send-comp-mbind-eq1*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ a ; *DONE-SEND*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ a ; *MAP-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-map-send6*)

lemma *map-send-comp-mbind-eq2*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ a ; *DONE-RECV*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ a ; *MAP-SEND*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-map-send7*)

lemma *buf-recv-comp-mbind-eq1*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ a ; *DONE-SEND*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-SEND*$_{MON}$ a ; *BUF-RECV*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-recv6*)

lemma *buf-recv-comp-mbind-eq2*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ a ; *DONE-RECV*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *DONE-RECV*$_{MON}$ a ; *BUF-RECV*$_{MON}$ a)) σ
by (*simp only*: *sem-comp-buf-recv7*)

lemma *buf-recv-comp-mbind-eq3*:
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *BUF-RECV*$_{MON}$ a ; *MAP-SEND*$_{MON}$ a)) σ =
mbind is ($\lambda a.$ (*out1* $\leftarrow$ *MAP-SEND*$_{MON}$ a ; *BUF-RECV*$_{MON}$ a)) σ

by (*simp only*: *sem-comp-buf-recv8*)

lemma *buf-recv-comp-mbind-eq4*:
mbind is (λ*a*. (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ =
mbind is (λ*a*. (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *BUF-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-buf-recv9*)

lemma *map-recv-comp-mbind-eq1*:
mbind is (λ*a*. (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-SEND*$_{MON}$ *a*)) σ =
mbind is (λ*a*. (*out1* ← *DONE-SEND*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-map-recv6*)

lemma *map-recv-comp-mbind-eq2*:
mbind is (λ*a*. (*out1* ← *MAP-RECV*$_{MON}$ *a* ; *DONE-RECV*$_{MON}$ *a*)) σ =
mbind is (λ*a*. (*out1* ← *DONE-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*)) σ
by (*simp only*: *sem-comp-map-recv7*)

end

theory *IPC-step-normalizer*

imports *IPC-traces*

begin

I IPC Stepping Function and Traces

definition
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* =
(*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)

definition
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ =
(¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ⟶
(*IPC-params-c2* ((*the o thread-list* σ) *partner*) ∧
IPC-params-c6 caller ((*the o thread-list* σ) *partner*)))

definition
exec-action$_{id}$*-Mon-prep-fact2 caller partner* σ =
(¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
IPC-params-c2 ((*the o thread-list* σ) *partner*)∧
¬*IPC-params-c6 caller* ((*the o thread-list* σ) *partner*))

definition
exec-action$_{id}$*-Mon-prep-send-fact3 caller error-mem* σ *msg* =
(¬(*list-all* ((*is-part-addr-th-mem o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)
∧

$error\text{-}mem = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}fact4\ caller\ partner\ error\text{-}mem\ \sigma\ msg =$
$((list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$\wedge$
$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ partner)\ (resource\ \sigma))msg) \wedge$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}fact3\ caller\ error\text{-}mem\ \sigma\ msg =$
$(\neg(list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$\wedge$
$error\text{-}mem = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}RECV)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}fact4\ caller\ partner\ error\text{-}mem\ \sigma\ msg =$
$((list\text{-}all\ ((is\text{-}part\text{-}addr\text{-}th\text{-}mem\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$\wedge$
$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ partner)\ (resource\ \sigma))msg)$
$\wedge$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV)$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact5\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\vee$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\wedge$
$IPC\text{-}params\text{-}c4\ \ caller\ partner\ ) \wedge$
$IPC\text{-}params\text{-}c3\ \ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\ )$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact6\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\vee$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\wedge$
$IPC\text{-}params\text{-}c4\ \ caller\ \ partner\ ) \wedge$
$\neg IPC\text{-}params\text{-}c3\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))$

definition
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact7\ caller\ partner\ \sigma =$
$(\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\vee$
$(IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\wedge$
$IPC\text{-}params\text{-}c4\ \ caller\ \ partner)\ )$

I.1 Simplification rules related to the stepping function $exec\text{-}action_{id}\text{-}Mon$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}mbind\text{-}obvious$:
$\bigwedge \sigma\ S.\ mbind\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by *simp*

lemma *exec-action$_{id}$-Mon-mbind-obvious'*:
(*case mbind S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *σ of*
None ⇒ *Some* ([*get-caller-error caller σ*], *σ*)
| *Some* (*outs*, *σ''*) ⇒ *a*) = *a*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *σ* = *Some a*
then show *?thesis*
using *hyp0*
by *simp*
qed

lemma *exec-action$_{id}$-Mon-all-obvious1*:
∀ *a σ*.∃ *errors σ'*. *exec-action$_{id}$-Mon a σ* = *Some* (*errors*, *σ'*)
by (*auto*, *rule action$_{ipc}$.induct*, *auto simp*:*exec-action$_{id}$-Mon-def*)

Simplification rules on *PREP* action

lemma *exec-action$_{id}$-Mon-prep-send-obvious0*:
⋀*σ*. *exec-action$_{id}$-Mon* (*IPC PREP* (*SEND caller partner msg*)) *σ* ≠ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-prep-send-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC PREP* (*SEND caller partner msg*)) *σ*) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list σ*) *caller*) (*resource σ*))*msg*)
then
if IPC-params-c1 ((*the o thread-list σ*) *partner*)
then Some (*NO-ERRORS*,
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list σ*),
error-codes := *NO-ERRORS*|))
else
if IPC-params-c2 ((*the o thread-list σ*) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list σ*) *partner*)
then Some (*NO-ERRORS*,
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list σ*),
error-codes := *NO-ERRORS*|))
else

Some(*ERROR-IPC error-IPC-22-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|))
else Some (*ERROR-IPC error-IPC-23-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|))

else Some (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list σ*),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|)))
by (*simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious2*:
(*fst o the*)(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) *σ*) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list σ*) *caller*) (*resource σ*))*msg*)
then
if IPC-params-c1 ((*the o thread-list σ*) *partner*)
then NO-ERRORS
else
(*if IPC-params-c2* ((*the o thread-list σ*) *partner*)
then
if IPC-params-c6 caller ((*the o thread-list σ*) *partner*)
then NO-ERRORS
else
ERROR-IPC error-IPC-22-in-PREP-SEND
else ERROR-IPC error-IPC-23-in-PREP-SEND)
else ERROR-MEM not-valid-sender-addr-in-PREP-SEND)
by (*simp add*:*exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious3*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) *σ* = *Some*(*NO-ERRORS*,
σ′)) =
(*σ′*= *σ*(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list σ*),
error-codes := *NO-ERRORS*|) ∧
exec-action$_{id}$*-Mon-prep-fact0 caller partner σ msg* ∧
exec-action$_{id}$*-Mon-prep-fact1 caller partner σ*)
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*

exec-action$_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious4*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-MEM*

error-mem, σ')) =
((σ′= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|)∧

¬(*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*) ∧
error-mem = *not-valid-sender-addr-in-PREP-SEND*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-send-obvious5*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC*
error-IPC, σ')) =
((σ′= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|)∧
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
IPC-params-c2 ((*the o thread-list* σ) *partner*) ∧
¬ *IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-22-in-PREP-SEND*) ∨
(σ′= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|)∧
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
¬ *IPC-params-c2* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-23-in-PREP-SEND*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def*
exec-action$_{id}$*-Mon-prep-fact0-def*
split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious0*:
∀ σ. *exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ ≠ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ) =
(*if* (*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)
then
if IPC-params-c1 ((*the o thread-list* σ) *partner*)
then Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,

```
                thread-list    := update-th-ready caller (thread-list σ),
                error-codes    := NO-ERRORS⦈))
        else
         (if IPC-params-c2 ((the o thread-list σ)  partner)
          then
            if IPC-params-c6 caller ((the o thread-list σ) partner)
            then Some(NO-ERRORS,
                    σ⦇current-thread := caller,
                      thread-list    := update-th-ready caller (thread-list σ),
                      error-codes    := NO-ERRORS⦈)
            else
               Some(ERROR-IPC error-IPC-22-in-PREP-RECV,
                       σ⦇current-thread := caller ,
                         thread-list    := update-th-current caller (thread-list σ),
                       error-codes    := ERROR-IPC error-IPC-22-in-PREP-RECV⦈)
          else Some(ERROR-IPC error-IPC-23-in-PREP-RECV,
                    σ⦇current-thread := caller ,
                      thread-list    := update-th-current caller (thread-list σ),
                     error-codes    := ERROR-IPC error-IPC-23-in-PREP-RECV⦈)))

    else Some (ERROR-MEM not-valid-receiver-addr-in-PREP-RECV,
              σ⦇current-thread := caller ,
                thread-list    := update-th-current caller (thread-list σ),
            error-codes   := ERROR-MEM not-valid-receiver-addr-in-PREP-RECV⦈)))
  by(simp add: exec-action_id-Mon-def PREP-RECV_id-def)



lemma exec-action_id-Mon-prep-recv-obvious2:
  fst(the(exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ)) =
   (if (list-all ((is-part-mem-th o the) ((thread-list σ) caller) (resource σ))msg)
    then
        if IPC-params-c1 ((the o thread-list σ)  partner)
        then NO-ERRORS
        else
         (if IPC-params-c2 ((the o thread-list σ)  partner)
          then
            if IPC-params-c6 caller ((the o thread-list σ) partner)
            then NO-ERRORS
            else
              ERROR-IPC error-IPC-22-in-PREP-RECV
          else ERROR-IPC error-IPC-23-in-PREP-RECV)
    else ERROR-MEM not-valid-receiver-addr-in-PREP-RECV)
  unfolding exec-action_id-Mon-def
  by (simp add: exec-action_id-Mon-def PREP-RECV_id-def)

lemma exec-action_id-Mon-prep-recv-obvious3:
 (exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ = Some(NO-ERRORS,
```

σ')) =
  (σ'= σ(|*current-thread* := *caller*,
      thread-list := *update-th-ready caller* (*thread-list* σ),
      error-codes := *NO-ERRORS*|) ∧
   exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg*∧
   exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ)
 by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-RECV*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
      exec-action$_{id}$*-Mon-prep-fact1-def*
    split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious4*:
 (*exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')) =
  ((σ'= σ(|*current-thread* := *caller* ,
      thread-list := *update-th-current caller* (*thread-list* σ),
     error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*|)∧

   ¬(*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*) ∧
    error-mem = *not-valid-receiver-addr-in-PREP-RECV*))
 by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-RECV*$_{id}$*-def*
    split: *errors.split split-if split-if-asm*)

lemma *exec-action*$_{id}$*-Mon-prep-recv-obvious5*:
 (*exec-action*$_{id}$*-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
  ((σ'=σ(|*current-thread* := *caller* ,
      thread-list := *update-th-current caller* (*thread-list* σ),
      error-codes := *ERROR-IPC error-IPC-22-in-PREP-RECV*|)∧
    exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
   ¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
    IPC-params-c2 ((*the o thread-list* σ) *partner*) ∧
   ¬*IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) ∧
    error-IPC = *error-IPC-22-in-PREP-RECV*) ∨
   (σ'= σ(|*current-thread* := *caller* ,
      thread-list := *update-th-current caller* (*thread-list* σ),
      error-codes := *ERROR-IPC error-IPC-23-in-PREP-RECV*|)∧
    exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
    ¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
    ¬ *IPC-params-c2* ((*the o thread-list* σ) *partner*) ∧
    error-IPC = *error-IPC-23-in-PREP-RECV*))
 by (*auto simp add*: *exec-action*$_{id}$*-Mon-def PREP-RECV*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def*
      exec-action$_{id}$*-Mon-prep-fact0-def*
    split: *errors.split split-if split-if-asm*)

Simplification rules on WAIT action

lemma *exec-action*$_{id}$*-Mon-wait-send-obvious0*:
 ⋀σ. *exec-action*$_{id}$*-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ ≠ *None*
 unfolding *exec-action*$_{id}$*-Mon-def*

by *simp*

definition
exec-action$_{id}$*-Mon-wait-send-upd caller* σ =
(*case* (*thread-list* σ) *caller of None* $\Rightarrow$
σ (|*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*|)
| *Some th* $\Rightarrow$ σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*|))

lemma *exec-action*$_{id}$*-Mon-wait-send-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ) =
(*if* ¬ *IPC-send-comm-check-st*$_{id}$ *caller partner* σ
then Some(*ERROR-IPC error-IPC-1-in-WAIT-SEND* ,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*|))
else
if ¬ *IPC-params-c4 caller partner*
then Some(*ERROR-IPC error-IPC-3-in-WAIT-SEND*,
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*|)
)
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* $\Rightarrow$
Some (*ERROR-IPC error-IPC-6-in-WAIT-SEND* ,
σ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*|))
| *Some th* $\Rightarrow$ *Some* (*ERROR-IPC error-IPC-5-in-WAIT-SEND* ,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*|)))

else
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*simp add*: *exec-action*$_{id}$*-Mon-def WAIT-SEND*$_{id}$*-def list.induct split*: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-send-obvious2*:
fst (*the*(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ)) =
(*if* ¬ *IPC-send-comm-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-WAIT-SEND
else
if ¬ *IPC-params-c4 caller partner*
then ERROR-IPC error-IPC-3-in-WAIT-SEND
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
ERROR-IPC error-IPC-6-in-WAIT-SEND
| *Some th* ⇒ *ERROR-IPC error-IPC-5-in-WAIT-SEND*)
else
NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def list.induct*
split: *option.split*)

lemma *exec-action$_{id}$-Mon-wait-send-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')) =
(σ' = σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) ∧
IPC-send-comm-check-st$_{id}$ caller partner σ ∧
IPC-params-c4 caller partner ∧
IPC-params-c5 partner σ)
by (*auto simp add*: *exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def split*: *option.split-asm*)

definition
update-state-wait-send-params5 σ *caller* =
(*case* (*thread-list* σ) *caller of None* ⇒
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-SEND*|)
| *Some th* ⇒ σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*|))

definition
update-state-wait-recv-params5 σ *caller* =
(*case* (*thread-list* σ) *caller of None* ⇒
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-RECV*|)
| *Some th* ⇒ σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*|))

lemma *exec-action$_{id}$-Mon-wait-send-obvious4*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
((¬ *IPC-send-comm-check-st$_{id}$ caller partner* σ ⟶
σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*|) ∧
error-IPC = *error-IPC-1-in-WAIT-SEND*) ∧
(*IPC-send-comm-check-st$_{id}$ caller partner* σ ⟶
((¬ *IPC-params-c4 caller partner* ⟶
σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*|) ∧
error-IPC = *error-IPC-3-in-WAIT-SEND*) ∧
(*IPC-params-c4 caller partner* ⟶
((¬ *IPC-params-c5 partner* σ ⟶
σ'= *update-state-wait-send-params5* σ *caller* ∧
error-codes (*update-state-wait-send-params5* σ *caller*) = *ERROR-IPC error-IPC*) ∧
¬ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-send-params5-def exec-action$_{id}$-Mon-def WAIT-SEND$_{id}$-def*
split: *split-if-asm option.split-asm*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious0*:
⋀σ. *exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ ≠ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-wait-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ) =
(*if* ¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ
then Some(*ERROR-IPC error-IPC-1-in-WAIT-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|))
else
if ¬ *IPC-params-c4 caller partner*
then Some(*ERROR-IPC error-IPC-3-in-WAIT-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|))
else

if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
Some(*ERROR-IPC error-IPC-6-in-WAIT-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-6-in-WAIT-RECV*|))
| *Some th* ⇒ *Some*(*ERROR-IPC error-IPC-5-in-WAIT-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*|)))

else
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*simp add*: *exec-action*$_{id}$*-Mon-def* *WAIT-RECV*$_{id}$*-def list.induct split*: *option.split*)

lemma *exec-action*$_{id}$*-Mon-wait-recv-obvious2*:
fst(*the*(*exec-action*$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ)) =
(*if* ¬ *IPC-recv-comm-check-st*$_{id}$ *caller partner* σ
then ERROR-IPC error-IPC-1-in-WAIT-RECV
else
if ¬ *IPC-params-c4* *caller* *partner*
then ERROR-IPC error-IPC-3-in-WAIT-RECV
else
if ¬ *IPC-params-c5 partner* σ
then
(*case* (*thread-list* σ) *caller of None* ⇒
ERROR-IPC error-IPC-6-in-WAIT-RECV
| *Some th* ⇒ *ERROR-IPC error-IPC-5-in-WAIT-RECV*)
else
NO-ERRORS)
by (*simp add*: *exec-action*$_{id}$*-Mon-def* *WAIT-RECV*$_{id}$*-def list.induct split*: *option.split*)

lemma *exec-action*$_{id}$*-Mon-wait-recv-obvious3*:
(*exec-action*$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')) =
(σ' = σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|) ∧
IPC-recv-comm-check-st$_{id}$ *caller partner* σ ∧
IPC-params-c4 *caller* *partner* ∧
IPC-params-c5 partner σ)

by (*auto simp add*: *exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def split*: *list.split-asm*)

lemma *exec-action$_{id}$-Mon-wait-recv-obvious4*:
(*exec-action$_{id}$-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')) =
((¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ ⟶
σ'=σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|) ∧
error-IPC = *error-IPC-1-in-WAIT-RECV*) ∧
(*IPC-recv-comm-check-st$_{id}$ caller partner* σ ⟶
((¬ *IPC-params-c4 caller partner* ⟶
σ'= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|) ∧
error-IPC = *error-IPC-3-in-WAIT-RECV*) ∧
(*IPC-params-c4 caller partner* ⟶
((¬ *IPC-params-c5 partner* σ ⟶
σ'= *update-state-wait-recv-params5* σ *caller* ∧
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) ∧
¬ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-recv-params5-def exec-action$_{id}$-Mon-def WAIT-RECV$_{id}$-def*
split: *split-if-asm list.split-asm*)

Simplification rules on BUF action

lemma *exec-action$_{id}$-Mon-buf-send-obvious0*:
⋀σ. *exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) σ ≠ *None*
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-buf-send-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*SEND caller partner msg*)) σ) =
(*if* ¬ *IPC-buf-check-st$_{id}$ caller partner* σ
then Some (*ERROR-IPC error-IPC-1-in-BUF-SEND*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*|))
else
Some(*NO-ERRORS*,
σ(|*current-thread* := *caller*,
resource := *foldl* (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs partner* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*

$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\rparloor))$
by $(simp\ add\colon exec\text{-}action_{id}\text{-}Mon\text{-}def\ BUF\text{-}SEND_{id}\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}send\text{-}obvious2\colon$
$fst\ (the(exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma)) =$
$(if\ \neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma$
$then\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND$
$else\ NO\text{-}ERRORS)$
by $(simp\ add\colon exec\text{-}action_{id}\text{-}Mon\text{-}def\ BUF\text{-}SEND_{id}\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}send\text{-}obvious3\colon$
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some(error, \sigma'))$
$=$
$((\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$\sigma' = \sigma\lparloor current\text{-}thread := caller\ ,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\rparloor \wedge$
$error = ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND) \wedge$
$((IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$(\sigma' = \sigma\lparloor current\text{-}thread := caller,$
$resource\ := foldl\ (\lambda m\ (addr,val).\ (m\ (addr :=_{\$}\ val)))\ (resource\ \sigma)$
$(zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\rparloor \wedge$
$error = NO\text{-}ERRORS)) \vee$
$(IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$(msg = [] \wedge$
$\sigma' = \sigma\lparloor current\text{-}thread := caller,$
$resource\ := resource\ \sigma,$
$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS\rparloor \wedge$
$error = NO\text{-}ERRORS))))$
by $(auto\ simp\ add\colon exec\text{-}action_{id}\text{-}Mon\text{-}def\ BUF\text{-}SEND_{id}\text{-}def)$

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}recv\text{-}obvious0\colon$
$\forall \sigma.\ exec\text{-}action_{id}\text{-}Mon\ \ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by $simp$

lemma *exec-action$_{id}$-Mon-buf-recv-obvious1*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ) =
(*if* $\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ
then Some (*ERROR-IPC error-IPC-1-in-BUF-RECV*,
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|))
else
Some(*NO-ERRORS*,
(σ(|*current-thread* := *caller*,
resource :=*foldl* (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))))
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ)) =
(*if* $\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ
then ERROR-IPC error-IPC-1-in-BUF-RECV
else NO-ERRORS)
by (*simp add*: *exec-action$_{id}$-Mon-def BUF-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-buf-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*error*, σ')) =
((¬ *IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
σ' = σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|) $\wedge$
error = *ERROR-IPC error-IPC-1-in-BUF-RECV*) $\wedge$
((*IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$
(σ' = σ(|*current-thread* := *caller*,
resource :=
foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|) $\wedge$
error =*NO-ERRORS*)) $\vee$
(*IPC-buf-check-st$_{id}$ caller partner* σ $\longrightarrow$

$(msg = [] \wedge$
$\sigma' = \sigma(\!|current\text{-}thread := caller,$
$resource := resource\ \sigma,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \wedge$
$error = NO\text{-}ERRORS))))$
by (*auto simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def\ BUF\text{-}RECV_{id}\text{-}def$)

Simplification rules on MAP action

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious0$:
$\bigwedge\sigma.\ exec\text{-}action_{id}\text{-}Mon\ \ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma \neq None$
unfolding $exec\text{-}action_{id}\text{-}Mon\text{-}def$
by *simp*

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious1$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma) =$
$Some(NO\text{-}ERRORS,$
$\sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ (resource\ \sigma)$
$(zip\ msg\ (get\text{-}th\text{-}addrs\ partner\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS|\!))$
by (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def\ MAP\text{-}SEND_{id}\text{-}def$)

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious2$:
$fst\ (the(exec\text{-}action_{id}\text{-}Mon\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma)) = NO\text{-}ERRORS$
by (*simp add*: $exec\text{-}action_{id}\text{-}Mon\text{-}def\ MAP\text{-}SEND_{id}\text{-}def$)

lemma $exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious3$:
$(exec\text{-}action_{id}\text{-}Mon\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(error,$
$\sigma')) =$
$((\sigma' = \sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ (resource\ \sigma)$
$(zip\ msg\ (get\text{-}th\text{-}addrs\ partner\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \wedge$
$error = NO\text{-}ERRORS) \vee$
$(msg = [] \wedge$

$\sigma' = \sigma(\!|$*current-thread* := *caller*,
resource := *resource* σ,
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*$|\!)$ $\wedge$
error =*NO-ERRORS*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def MAP-SEND*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-map-recv-obvious0*:
$\bigwedge\sigma$. *exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-map-recv-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ) =
Some(*NO-ERRORS*,
$\sigma(\!|$*current-thread* := *caller*,
resource := *foldl* (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*$|\!)$)
by (*simp add*: *exec-action*$_{id}$*-Mon-def MAP-RECV*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-map-recv-obvious2*:
fst (*the*(*exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ)) = *NO-ERRORS*
by (*simp add*: *exec-action*$_{id}$*-Mon-def MAP-RECV*$_{id}$*-def*)

lemma *exec-action*$_{id}$*-Mon-map-recv-obvious3*:
(*exec-action*$_{id}$*-Mon* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*error*,
σ')) =
(($\sigma' = \sigma(\!|$*current-thread* := *caller*,
resource := *foldl* (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*$|\!)$ $\wedge$
error =*NO-ERRORS*) $\vee$
(*msg* = [] $\wedge$
$\sigma' = \sigma(\!|$*current-thread* := *caller*,
resource := (*resource* σ),

thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*⦈ ∧
error =*NO-ERRORS*))
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def MAP-RECV*$_{id}$*-def*)

Simplification rules on DONE action

lemma *exec-action*$_{id}$*-Mon-done-send-obvious0*:
∀σ. *exec-action*$_{id}$*-Mon* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-done-send-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC DONE* (*SEND caller partner msg*)) σ) =
Some(*error-codes* σ, σ)
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-done-send-obvious2*:
fst (*the*(*exec-action*$_{id}$*-Mon* (*IPC DONE* (*SEND caller partner msg*)) σ)) =
error-codes σ
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-done-send-obvious3*:
(*exec-action*$_{id}$*-Mon* (*IPC DONE* (*SEND caller partner msg*)) σ = *Some*(*error*, σ')) =
(σ'= σ ∧ *error-codes* σ = *error*)
by (*auto simp add*: *exec-action*$_{id}$*-Mon-def*)

lemma *exec-action*$_{id}$*-Mon-done-recv-obvious0*:
⋀σ. *exec-action*$_{id}$*-Mon* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*
unfolding *exec-action*$_{id}$*-Mon-def*
by *simp*

lemma *exec-action*$_{id}$*-Mon-done-recv-obvious1*:
(*exec-action*$_{id}$*-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ) =
Some(*error-codes* σ, σ)
unfolding *exec-action*$_{id}$*-Mon-def*

by *simp*

lemma *exec-action$_{id}$-Mon-done-recv-obvious2*:
fst(*the*(*exec-action$_{id}$-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ)) =
error-codes σ
unfolding *exec-action$_{id}$-Mon-def*
by *simp*

lemma *exec-action$_{id}$-Mon-done-recv-obvious3*:
(*exec-action$_{id}$-Mon* (*IPC DONE* (*RECV caller partner msg*)) σ = *Some*(*error*, σ')) =
($\sigma' = \sigma \wedge$ *error-codes* σ = *error*)
by (*auto simp add*: *exec-action$_{id}$-Mon-def*)

lemma *exec-action$_{id}$-Mon-act-info-obvious0*:
(*exec-action$_{id}$-Mon a* σ = *Some*(*error*, σ')) $\Longrightarrow$
(*state$_{id}$.th-flag* σ = *state$_{id}$.th-flag* σ')
unfolding *exec-action$_{id}$-Mon-def*
by (*auto*, *rule action$_{ipc}$.induct* , *rule p4-stage$_{ipc}$.induct*,*rule p4-direct$_{ipc}$.induct*,
auto, *rule action$_{ipc}$.induct*, *simp-all*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
auto simp: *PREP-SEND$_{id}$-def PREP-RECV$_{id}$-def*,*rule p4-direct$_{ipc}$.induct*,
auto,
simp add: *WAIT-SEND$_{id}$-def split*: *option.split*, *simp add*: *WAIT-RECV$_{id}$-def*
split: *option.split*,
rule p4-direct$_{ipc}$.induct, *auto simp add*: *BUF-SEND$_{id}$-def BUF-RECV$_{id}$-def*,

rule p4-direct$_{ipc}$.induct,*auto simp add*: *MAP-SEND$_{id}$-def MAP-RECV$_{id}$-def*,

rule p4-direct$_{ipc}$.induct, *auto*)

lemma *exec-action$_{id}$-Mon-act-info-obvious0'*:
(*exec-action$_{id}$-Mon a* σ = *Some*(*error*, σ')) =
(*state$_{id}$.th-flag* σ = *state$_{id}$.th-flag* $\sigma' \wedge$ *error-codes* (*exec-action$_{id}$* σ *a*) = *error*
$\wedge$
exec-action$_{id}$ σ *a* = σ')
unfolding *exec-action$_{id}$-Mon-def*
by (*auto*, *rule action$_{ipc}$.induct* , *rule p4-stage$_{ipc}$.induct*,*rule p4-direct$_{ipc}$.induct*,
auto, *rule action$_{ipc}$.induct*, *simp-all*, *rule p4-stage$_{ipc}$.induct*, *rule p4-direct$_{ipc}$.induct*,
auto simp: *PREP-SEND$_{id}$-def PREP-RECV$_{id}$-def*,*rule p4-direct$_{ipc}$.induct*,
auto,
simp add: *WAIT-SEND$_{id}$-def split*: *option.split*, *simp add*: *WAIT-RECV$_{id}$-def*
split: *option.split*,
rule p4-direct$_{ipc}$.induct, *auto simp add*: *BUF-SEND$_{id}$-def BUF-RECV$_{id}$-def*,

rule p4-direct$_{ipc}$.induct,auto simp add: *MAP-SEND$_{id}$-def MAP-RECV$_{id}$-def*,

rule p4-direct$_{ipc}$.induct, *auto*)

lemma *exec-action$_{id}$-Mon-act-info-obvious1*:
exec-action$_{id}$-Mon (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*error*, σ') $\Longrightarrow$
(*state$_{id}$.th-flag* σ) *caller* = (*state$_{id}$.th-flag* σ') *caller*
by (*auto simp*:*exec-action$_{id}$-Mon-def PREP-RECV$_{id}$-def*)

lemma *exec-action$_{id}$-Mon-act-info-obvious2*:
(*state$_{id}$.th-flag* σ) *caller* =
(*th-flag*(*snd*(*the*(*exec-action$_{id}$-Mon* (*IPC PREP* (*RECV caller partner msg*)) σ)))) *caller*

unfolding *exec-action$_{id}$-Mon-def*
by (*simp add*: *PREP-RECV$_{id}$-def*)

lemma *exec-errors-obvious0*: (*exec-action$_{id}$-Mon a* σ) = *Some* (*NO-ERRORS*,σ') $\Longrightarrow$
error-codes σ' = *NO-ERRORS*
by (*auto simp only*: *exec-action$_{id}$-Mon-def prod.inject the.simps*)

lemma *exec-errors-obvious1*: (*exec-action$_{id}$-Mon a* σ) = *Some* (*NO-ERRORS*,σ') $\Longrightarrow$
error-codes $\sigma' \neq$ *ERROR-MEM error-mem*
by (*auto simp only*: *exec-action$_{id}$-Mon-def prod.inject the.simps*)

lemma *exec-errors-obvious2*: (*exec-action$_{id}$-Mon a* σ) = *Some* (*NO-ERRORS*,σ') $\Longrightarrow$
error-codes $\sigma' \neq$ *ERROR-IPC error-ipc*
by (*auto simp only*: *exec-action$_{id}$-Mon-def prod.inject the.simps*)

lemmas *step-normalizer-None* =
exec-action$_{id}$-Mon-prep-send-obvious0 exec-action$_{id}$-Mon-prep-recv-obvious0
exec-action$_{id}$-Mon-wait-send-obvious0 exec-action$_{id}$-Mon-wait-recv-obvious0
exec-action$_{id}$-Mon-buf-send-obvious0 exec-action$_{id}$-Mon-buf-recv-obvious0
exec-action$_{id}$-Mon-done-send-obvious0 exec-action$_{id}$-Mon-done-recv-obvious0

lemmas *step-normalizer-Some* = *exec-action$_{id}$-Mon-act-info-obvious0'*
end

theory *IPC-atomic-action-normalizer*

imports *IPC-step-normalizer*

begin

J Atomic Actions Reasoning

J.1 Symbolic Execution Rules of Atomic Actions

lemma *prep-send-obvious*:
$(PREP\text{-}SEND_{id}\ \sigma\ (IPC\ PREP\ (SEND\ caller\ partner\ msg)) = \sigma') =$
$(((\sigma' = \sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma)) \vee$

$(\ (\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$\wedge$
$\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND|\!)) \vee$

$((\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner))\ \vee$
$(\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner))))$
by $(auto\ simp\ add: PREP\text{-}SEND_{id}\text{-}def\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\text{-}def$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\text{-}def$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact2\text{-}def)$

lemma *wait-send-obvious*:
$(WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ =\ \sigma') =$
$(\sigma' = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \wedge$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \wedge$
$IPC\text{-}params\text{-}c4\ \ caller\ partner\ \wedge$
$IPC\text{-}params\text{-}c5\ partner\ \sigma) \vee$

$((\neg\ IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$

$\sigma' = \sigma$ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*|)) ∧
(*IPC-send-comm-check-st*$_{id}$ *caller partner* $\sigma \longrightarrow$
((¬ *IPC-params-c4 caller partner* $\longrightarrow$
$\sigma' = \sigma$ (|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-SEND*|)))))
by (*auto simp add*: *update-state-wait-send-params5-def WAIT-SEND*$_{id}$*-def*
split: *split-if-asm option.split-asm*)

lemma *buf-send-obvious*:
(*BUF-SEND*$_{id}$ σ (*IPC BUF* (*SEND caller partner msg*)) = σ') =
((¬ *IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \longrightarrow$
$\sigma' = \sigma$(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*|) ∧
(*IPC-buf-check-st*$_{id}$ *caller partner* $\sigma \longrightarrow$
($\sigma' = \sigma$(|*current-thread* := *caller*,
resource :=
foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs partner* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))))
by (*auto simp add*:*BUF-SEND*$_{id}$*-def*)

lemma *map-send-obvious*:
(*MAP-SEND*$_{id}$ σ (*IPC MAP* (*SEND caller partner msg*)) = σ') =
($\sigma' = \sigma$(|*current-thread* := *caller*,
resource :=
foldl (λ*m* (*src*,*dst*). (*m* (*src*⋈ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs partner* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: *MAP-SEND*$_{id}$*-def*)

lemma *prep-recv-obvious*:
(*PREP-RECV*$_{id}$ σ (*IPC PREP* (*RECV caller partner msg*)) =σ') =

$$
\begin{array}{l}
(((\sigma' = \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma)},\\
\qquad \mathit{error\text{-}codes} \quad := \mathit{NO\text{-}ERRORS}|\!) \wedge\\
\quad \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg} \wedge\\
\quad \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma}))\vee\\
\\
(\ (\neg(\mathit{list\text{-}all}\ ((\mathit{is\text{-}part\text{-}mem\text{-}th\ o\ the})\ ((\mathit{thread\text{-}list}\ \sigma)\ \mathit{caller})\ (\mathit{resource}\ \sigma))\mathit{msg})\\
\wedge\\
\quad \sigma' = \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ ,\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},\\
\quad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV}|\!)))\vee\\
\\
(\ (\sigma' = \ \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ ,\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},\\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV}|\!) \wedge\\
\quad \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg} \wedge\\
\quad \neg\mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the\ o\ thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \wedge\\
\quad \mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the\ o\ thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \wedge\\
\quad \neg\mathit{IPC\text{-}params\text{-}c6\ caller}\ ((\mathit{the\ o\ thread\text{-}list}\ \sigma)\ \mathit{partner}))\ \ \vee\\
\quad (\sigma' = \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ ,\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},\\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV}|\!) \wedge\\
\quad \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg} \wedge\\
\quad \neg\mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the\ o\ thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \wedge\\
\quad \neg\mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the\ o\ thread\text{-}list}\ \sigma)\ \ \mathit{partner}))\ ))\\
\textbf{by}\ (\mathit{auto\ simp\ add}\colon \mathit{PREP\text{-}RECV_{id}\text{-}def}\ \ \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact2\text{-}def}\\
\qquad \mathit{exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\text{-}def\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\text{-}def})
\end{array}
$$

lemma *wait-recv-obvious*:

$$
\begin{array}{l}
(\mathit{WAIT\text{-}RECV_{id}}\ \sigma\ (\mathit{IPC\ WAIT\ (RECV\ caller\ partner\ msg)})\ \ =\ \ \sigma') =\\
\quad (\sigma' = \sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ ,\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma)},\\
\qquad \mathit{error\text{-}codes} \quad := \mathit{NO\text{-}ERRORS}|\!) \wedge\\
\quad \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma} \wedge\\
\quad \mathit{IPC\text{-}params\text{-}c4}\ \ \mathit{caller\ partner}\ \wedge\\
\quad \mathit{IPC\text{-}params\text{-}c5\ partner\ \sigma})\ \vee\\
\\
\quad ((\neg\ \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma} \longrightarrow\\
\qquad \sigma' = \sigma\ (\!|\mathit{current\text{-}thread} := \mathit{caller},\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},\\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}|\!)\ ) \wedge\\
\quad (\mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma} \longrightarrow\\
\quad ((\neg\ \mathit{IPC\text{-}params\text{-}c4\ caller\ partner} \longrightarrow\\
\qquad \sigma' = \sigma\ (\!|\mathit{current\text{-}thread} := \mathit{caller},\\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},
\end{array}
$$

error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*⦈)))))
by (*auto simp add*: *update-state-wait-recv-params5-def* *WAIT-RECV*$_{id}$*-def*
split: *split-if-asm*)

lemma *buf-recv-obvious*:
(*BUF-RECV*$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) = σ') =
((¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ ⟶
σ' = σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*⦈) ∧
(*IPC-buf-check-st*$_{id}$ *caller partner* σ ⟶
(σ' = σ⦇*current-thread* := *caller*,
resource :=
foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*⦈)))
by (*auto simp add*: *BUF-RECV*$_{id}$*-def*)

lemma *map-recv-obvious*:
(*MAP-RECV*$_{id}$ σ (*IPC MAP* (*RECV caller partner msg*)) = σ') =
(σ' = σ⦇*current-thread* := *caller*,
resource :=
foldl (λ*m* (*src*,*dst*). (*m* (*src*⋈ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*⦈)
by (*auto simp add*: *MAP-RECV*$_{id}$*-def*)

J.2 Symbolic Execution Rules for Error Codes Field

lemma *PREP-SEND*$_{id}$*-obvious0*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) =
NO-ERRORS) =
(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg*∧
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ ∧
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ⦇*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈))

by (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
exec-action$_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)

lemma *PREP-SEND*$_{id}$*-obvious1*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) =
ERROR-MEM error-mem) =
(¬((*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*))
∧
error-mem = *not-valid-sender-addr-in-PREP-SEND* ∧
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|)))
by (*auto simp add*: *PREP-SEND*$_{id}$*-def split*: *errors.split split-if split-if-asm*)

lemma *PREP-SEND*$_{id}$*-obvious2*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) =
ERROR-IPC error-IPC) =
(¬(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
IPC-params-c2 ((*the o thread-list* σ) *partner*) ∧
¬*IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-22-in-PREP-SEND* ∧
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-22-in-PREP-SEND*|)) ⟶
(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* ∧
¬*IPC-params-c1* ((*the o thread-list* σ) *partner*) ∧
¬*IPC-params-c2* ((*the o thread-list* σ) *partner*) ∧
error-IPC = *error-IPC-23-in-PREP-SEND* ∧
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-23-in-PREP-SEND*|)))
)
by (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact2-def exec-action*$_{id}$*-Mon-prep-fact1-def*
exec-action$_{id}$*-Mon-prep-fact0-def*
split: *errors.split split-if split-if-asm*)

lemma *WAIT-SEND*$_{id}$*-obvious0*:
(*error-codes* (*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*))) =
NO-ERRORS) =
(*IPC-send-comm-check-st*$_{id}$ *caller partner* σ ∧
IPC-params-c4 caller partner ∧
IPC-params-c5 partner σ ∧

$(WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list \quad := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := NO\text{-}ERRORS|\!|))$
by (*auto simp add*: $WAIT\text{-}SEND_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma $WAIT\text{-}SEND_{id}$*-obvious1*:
$(error\text{-}codes\ (WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))) =$
$ERROR\text{-}IPC\ error\text{-}IPC) =$
$((\neg\ IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$(WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg)))=$
$\sigma\ (\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND|\!|\ \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND)\ \wedge$
$(IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\neg\ IPC\text{-}params\text{-}c4\ \ caller\ \ partner \longrightarrow$
$(WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg)))=$
$\sigma\ (\!|current\text{-}thread := caller,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND|\!|\ \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND)\ \wedge$
$(IPC\text{-}params\text{-}c4\ \ caller\ partner \longrightarrow$
$((\neg\ IPC\text{-}params\text{-}c5\ partner\ \sigma \longrightarrow$
$(WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg)))=$
$update\text{-}state\text{-}wait\text{-}send\text{-}params5\ \sigma\ caller\ \wedge$
$error\text{-}codes\ (update\text{-}state\text{-}wait\text{-}send\text{-}params5\ \sigma\ caller) = ERROR\text{-}IPC$
$error\text{-}IPC)\ \wedge$
$\neg\ IPC\text{-}params\text{-}c5\ partner\ \sigma)))))$
by (*auto simp add*: *update-state-wait-send-params5-def* $WAIT\text{-}SEND_{id}$*-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma $WAIT\text{-}SEND_{id}$*-obvious2*:
$\neg(error\text{-}codes\ (WAIT\text{-}SEND_{id}\ \sigma\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))) =$
$ERROR\text{-}MEM\ error\text{-}IPC)$
by (*auto simp add*:$WAIT\text{-}SEND_{id}$*-def split*: *errors.split split-if split-if-asm option.split-asm*)

lemma $BUF\text{-}SEND_{id}$*-obvious0*:
$(error\text{-}codes\ (BUF\text{-}SEND_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))) =$
$NO\text{-}ERRORS) =$
$(IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \wedge$
$BUF\text{-}SEND_{id}\ \sigma\ (IPC\ BUF\ (SEND\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller,$
$resource \quad :=$

foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs partner* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: *BUF-SEND*$_{id}$*-def*)

lemma *BUF-SEND*$_{id}$*-obvious1*:
(*error-codes* (*BUF-SEND*$_{id}$ σ (*IPC BUF* (*SEND caller partner msg*))) =
ERROR-IPC error-IPC-1-in-BUF-SEND) =
(¬ *IPC-buf-check-st*$_{id}$ *caller partner* σ ∧
BUF-SEND$_{id}$ σ (*IPC BUF* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-SEND*|))
by (*auto simp add*: *BUF-SEND*$_{id}$*-def*)

lemma *MAP-SEND*$_{id}$*-obvious0*:
(*error-codes* (*MAP-SEND*$_{id}$ σ (*IPC MAP* (*SEND caller partner msg*))) =
error) =
(*error* = *NO-ERRORS* ∧
MAP-SEND$_{id}$ σ (*IPC MAP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller*,
resource := *foldl* (λ*m* (*src*,*dst*). (*m* (*src*⋈ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs partner* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: *MAP-SEND*$_{id}$*-def*)

lemma *DONE-SEND*$_{id}$*-obvious0*:
(*error-codes* (*exec-action*$_{id}$ σ (*IPC DONE* (*SEND caller partner msg*))) =
error) =
((*exec-action*$_{id}$ σ (*IPC DONE* (*SEND caller partner msg*)))= σ ∧ *error-codes*
σ = *error*)
by *simp*

lemma *PREP-RECV*$_{id}$*-obvious0*:
(*error-codes* (*PREP-RECV*$_{id}$ σ (*IPC PREP* (*RECV caller partner msg*))) =
NO-ERRORS) =
(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg*∧

$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma\ \wedge$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|\!|)$
)
by (*auto simp add*: $PREP\text{-}RECV_{id}$*-def* $exec\text{-}action_{id}$*-Mon-prep-fact0-def*
$exec\text{-}action_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)

lemma $PREP\text{-}RECV_{id}$*-obvious1*:
$(error\text{-}codes\ (PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))) =$
$ERROR\text{-}MEM\ error\text{-}mem) =$
$(\neg((list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg))$
$\wedge$
$error\text{-}mem = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ \wedge$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV|\!|)))$
by (*auto simp add*: $PREP\text{-}RECV_{id}$*-def split*: *errors.split split-if split-if-asm*)

lemma $PREP\text{-}RECV_{id}$*-obvious2*:
$(error\text{-}codes\ (PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))) =$
$ERROR\text{-}IPC\ error\text{-}IPC) =$
$(\neg(\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner)\ \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\ \wedge$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV|\!|)))$
$\longrightarrow$
$(exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg\ \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ \wedge$
$(PREP\text{-}RECV_{id}\ \sigma\ (IPC\ PREP\ (RECV\ caller\ partner\ msg)) =$
$\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV|\!|)))$

)
by (*auto simp add*: $PREP\text{-}RECV_{id}$*-def* $exec\text{-}action_{id}$*-Mon-prep-fact0-def*
$exec\text{-}action_{id}$*-Mon-prep-fact1-def* $exec\text{-}action_{id}$*-Mon-prep-fact2-def*
split: *errors.split split-if split-if-asm*)

lemma *WAIT-RECV$_{id}$-obvious0*:
(*error-codes* (*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*))) = *NO-ERRORS*) =
(*IPC-recv-comm-check-st$_{id}$ caller partner* σ $\wedge$
IPC-params-c4 caller partner $\wedge$
IPC-params-c5 partner σ $\wedge$
(*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)))
by (*auto simp add*: *WAIT-RECV$_{id}$-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV$_{id}$-obvious1*:
(*error-codes* (*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*))) = *ERROR-IPC error-IPC*) =
((¬ *IPC-recv-comm-check-st$_{id}$ caller partner* σ $\longrightarrow$
(*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*)))=
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|) $\wedge$
error-IPC = *error-IPC-1-in-WAIT-RECV*) $\wedge$
(*IPC-recv-comm-check-st$_{id}$ caller partner* σ $\longrightarrow$
((¬ *IPC-params-c4 caller partner* $\longrightarrow$
(*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*)))=
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|) $\wedge$
error-IPC = *error-IPC-3-in-WAIT-RECV*) $\wedge$
(¬ ¬ *IPC-params-c4 caller partner* $\longrightarrow$
((¬ *IPC-params-c5 partner* σ $\longrightarrow$
(*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*)))=
update-state-wait-recv-params5 σ *caller* $\wedge$
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC* $\wedge$
¬ *IPC-params-c5 partner* σ)))))
by (*auto simp add*: *update-state-wait-recv-params5-def WAIT-RECV$_{id}$-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV$_{id}$-obvious2*:
¬(*error-codes* (*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*))) = *ERROR-MEM error-mem*)
by (*auto simp add*: *WAIT-RECV$_{id}$-def*
split: *errors.split split-if split-if-asm option.split-asm*)

lemma *BUF-RECV$_{id}$-obvious0*:
(*error-codes* (*BUF-RECV$_{id}$* σ (*IPC BUF* (*RECV caller partner msg*))) = *NO-ERRORS*) =
(*IPC-buf-check-st$_{id}$ caller partner* σ $\wedge$
BUF-RECV$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller*,
resource :=
foldl (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: *BUF-RECV$_{id}$-def*)

lemma *BUF-RECV$_{id}$-obvious1*:
(*error-codes* (*BUF-RECV$_{id}$* σ (*IPC BUF* (*RECV caller partner msg*))) =
ERROR-IPC error-IPC-1-in-BUF-RECV) =
($\neg$ *IPC-buf-check-st$_{id}$ caller partner* σ $\wedge$
BUF-RECV$_{id}$ σ (*IPC BUF* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*|))
by (*auto simp add*: *BUF-RECV$_{id}$-def*)

lemma *MAP-RECV$_{id}$-obvious0*:
(*error-codes* (*MAP-RECV$_{id}$* σ (*IPC MAP* (*RECV caller partner msg*))) = *error*)
=
(*error* = *NO-ERRORS* $\wedge$
MAP-RECV$_{id}$ σ (*IPC MAP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller*,
resource :=
foldl (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|))
by (*auto simp add*: *MAP-RECV$_{id}$-def*)

lemma *DONE-RECV$_{id}$-obvious0*:
(*error-codes* (*exec-action$_{id}$* σ (*IPC DONE* (*RECV caller partner msg*))) =
error) =

((*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*)))= σ $\wedge$ *error-codes* σ = *error*)
by *simp*

J.3 Symbolic Execution Rules for Error Codes field on Pure-level

lemma *PREP-SEND*$_{id}$*-Pure-obvious0*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *NO-ERRORS* $\Longrightarrow$ *P*) $\equiv$
(*exec-action*$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* &&&
exec-action$_{id}$*-Mon-prep-fact1 caller partner* σ &&&
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|)$\Longrightarrow$ *P*)
find-theorems *name*:*Pure*.
apply (*rule equal-intr-rule*)
apply (*elim meta-impE*)
apply (*drule conjunctionD2*)
apply (*drule conjunctionD2*)
apply (*auto simp add*: *PREP-SEND*$_{id}$*-def exec-action*$_{id}$*-Mon-prep-fact0-def*
exec-action$_{id}$*-Mon-prep-fact1-def*
split: *errors.split split-if split-if-asm*)
done

lemma *PREP-SEND*$_{id}$*-Pure-obvious1*:
(*error-codes* (*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*))) = *ERROR-MEM error-mem* $\Longrightarrow$*P*) $\equiv$
(¬((*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*)) &&&
error-mem = *not-valid-sender-addr-in-PREP-SEND* &&&
(*PREP-SEND*$_{id}$ σ (*IPC PREP* (*SEND caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*|))
$\Longrightarrow$ *P*
)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*: *PREP-SEND*$_{id}$*-def split*: *errors.split split-if split-if-asm*)

lemma *WAIT-SEND*$_{id}$*-Pure-obvious0*:
(*error-codes* (*WAIT-SEND*$_{id}$ σ (*IPC WAIT* (*SEND caller partner msg*))) = *NO-ERRORS* $\Longrightarrow$ *P*) $\equiv$
(*IPC-send-comm-check-st*$_{id}$ *caller partner* σ &&&
IPC-params-c4 caller partner &&&

IPC-params-c5 partner σ &&&
(WAIT-SEND$_{id}$ σ (IPC WAIT (SEND caller partner msg)) =
σ(|current-thread := caller ,
thread-list := update-th-waiting caller (thread-list σ),
error-codes := NO-ERRORS|))
⟹ P)
apply *(rule equal-intr-rule)*
apply *(drule conjunctionD2)+*
by *(auto simp add:WAIT-SEND$_{id}$-def split: errors.split split-if split-if-asm option.split-asm)*

lemma *WAIT-SEND$_{id}$-Pure-obvious1:*
(error-codes (WAIT-SEND$_{id}$ σ (IPC WAIT (SEND caller partner msg))) = ERROR-IPC error-IPC ⟹ P) ≡
((¬ IPC-send-comm-check-st$_{id}$ caller partner σ ⟹
(WAIT-SEND$_{id}$ σ (IPC WAIT (SEND caller partner msg)))=
σ (|current-thread := caller,
thread-list := update-th-current caller (thread-list σ),
error-codes := ERROR-IPC error-IPC-1-in-WAIT-SEND|) &&&
error-IPC = error-IPC-1-in-WAIT-SEND) &&&
(IPC-send-comm-check-st$_{id}$ caller partner σ ⟹
((¬ IPC-params-c4 caller partner ⟹
(WAIT-SEND$_{id}$ σ (IPC WAIT (SEND caller partner msg)))=
σ (|current-thread := caller,
thread-list := update-th-current caller (thread-list σ),
error-codes := ERROR-IPC error-IPC-3-in-WAIT-SEND|) &&&
error-IPC = error-IPC-3-in-WAIT-SEND) &&&
(IPC-params-c4 caller partner ⟹
((¬ IPC-params-c5 partner σ ⟹
(WAIT-SEND$_{id}$ σ (IPC WAIT (SEND caller partner msg)))= update-state-wait-send-params5 σ caller &&&
error-codes (update-state-wait-send-params5 σ caller) = ERROR-IPC error-IPC) &&&
¬ IPC-params-c5 partner σ))))
⟹ P)
apply *(rule equal-intr-rule)*
apply *(simp-all add: conjunction-imp Pure.imp-conjunction)*
by *(simp-all add: update-state-wait-send-params5-def WAIT-SEND$_{id}$-def split: errors.split split-if split-if-asm option.split option.split-asm)*

lemma *DONE-SEND$_{id}$-Pure-obvious0:*
(error-codes (exec-action$_{id}$ σ (IPC DONE (SEND caller partner msg))) = error ⟹P) ≡
((exec-action$_{id}$ σ (IPC DONE (SEND caller partner msg)))= σ ⟹ error-codes σ = error ⟹P)

by *simp*

lemma *PREP-RECV$_{id}$-Pure-obvious0*:
(*error-codes* (*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*))) =
NO-ERRORS $\Longrightarrow$ *P*) $\equiv$
(*exec-action$_{id}$-Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
exec-action$_{id}$-Mon-prep-fact1 caller partner σ $\Longrightarrow$
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
$\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
by (*auto simp add*: *PREP-RECV$_{id}$-def exec-action$_{id}$-Mon-prep-fact0-def*
exec-action$_{id}$-Mon-prep-fact1-def
split: *errors.split split-if split-if-asm*)

lemma *PREP-RECV$_{id}$-Pure-obvious1*:
(*error-codes* (*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*))) =
ERROR-MEM error-mem $\Longrightarrow$ *P*) $\equiv$
(¬((*list-all* ((*is-part-mem-th o the*) ((*thread-list* σ) *caller*) (*resource* σ))*msg*))
&&&
error-mem = *not-valid-receiver-addr-in-PREP-RECV* &&&
(*PREP-RECV$_{id}$* σ (*IPC PREP* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-receiver-addr-in-PREP-RECV*|))

$\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*: *PREP-RECV$_{id}$-def split*: *errors.split split-if split-if-asm*)

lemma *WAIT-RECV$_{id}$-Pure-obvious0*:
(*error-codes* (*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*))) =
NO-ERRORS $\Longrightarrow$ *P*) $\equiv$
(*IPC-recv-comm-check-st$_{id}$ caller partner* σ &&&
IPC-params-c4 caller partner &&&
IPC-params-c5 partner σ &&&
(*WAIT-RECV$_{id}$* σ (*IPC WAIT* (*RECV caller partner msg*)) =
σ(|*current-thread* := *caller* ,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*|))
$\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)

apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*auto simp add*:*WAIT-RECV*$_{id}$*-def split*: *errors.split split-if split-if-asm option.split-asm*)

lemma *WAIT-RECV*$_{id}$*-Pure-obvious1*:
(*error-codes* (*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*))) = *ERROR-IPC error-IPC* $\Longrightarrow$ *P*)≡
((¬ *IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))=
σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*|)
&&&
error-IPC = *error-IPC-1-in-WAIT-RECV*) &&&
(*IPC-recv-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
((¬ *IPC-params-c4 caller partner* $\Longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))= σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*|)
&&&
error-IPC = *error-IPC-3-in-WAIT-RECV*) &&&
(*IPC-params-c4 caller partner* $\Longrightarrow$
((¬ *IPC-params-c5 partner* σ $\Longrightarrow$
(*WAIT-RECV*$_{id}$ σ (*IPC WAIT* (*RECV caller partner msg*)))= *update-state-wait-recv-params5* σ *caller* &&&
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) &&&
¬ *IPC-params-c5 partner* σ)))) $\Longrightarrow$ *P*)
apply (*rule equal-intr-rule*)
apply (*simp-all add*: *conjunction-imp Pure.imp-conjunction*)
by (*simp-all add*: *update-state-wait-recv-params5-def WAIT-RECV*$_{id}$*-def*
split: *errors.split split-if split-if-asm list.split-asm*)

lemma *DONE-RECV*$_{id}$*-Pure-obvious0*:
(*error-codes* (*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*))) = *error* $\Longrightarrow$ *P*) ≡
((*exec-action*$_{id}$ σ (*IPC DONE* (*RECV caller partner msg*)))= σ $\Longrightarrow$ *error-codes* σ = *error* $\Longrightarrow$ *P*)
by *simp*

J.4 Symbolic Execution of Action Informations Field

lemma *act-info-obvious0*:
((*th-flag* (*update-state caller* σ *f error*)) =
(*th-flag* σ)(*caller* := *None*)) =
((*state*$_{id}$.*th-flag* σ) = (*state*$_{id}$.*th-flag* σ)(*caller* := *None*))
by *simp*

lemma *act-info-obvious1*:
(*th-flag* (*update-state caller* (*init-act-info caller partner* σ) *f error*)) =
((*th-flag* σ) (*caller* := *None*, *partner* := *None*))
by *simp*

lemma *act-info-obvious2*:
(*th-flag* (*update-state caller* (*remove-caller-error caller* σ) *f error*)) =
((*th-flag* σ) (*caller* := *None*))
by *simp*

lemma *act-info-prep-send-obvious0*:
(*th-flag* (*PREP-SEND*$_{id}$ (*init-act-info caller partner* σ)
(*IPC PREP* (*SEND caller partner msg*)))) =
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*)
by (*simp add*: *PREP-SEND*$_{id}$*-def*)

lemma *act-info-prep-send-obvious1*:
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*) =
(*th-flag*(*PREP-SEND*$_{id}$(*init-act-info caller partner*
σ(|*current-thread* := *caller*,
thread-list := *th-list*,
error-codes := *error*|))
(*IPC PREP* (*SEND caller partner msg*))))
by (*simp add*: *PREP-SEND*$_{id}$*-def*)

lemma *act-info-wait-send-obvious0*:
(*th-flag* (*WAIT-SEND*$_{id}$ (*init-act-info caller partner* σ)
(*IPC WAIT* (*SEND caller partner msg*)))) =
(*th-flag* σ)(*caller* := *None*, *partner* := *None*)
by (*simp add*: *WAIT-SEND*$_{id}$*-def split*: *option.split*)

lemma *act-info-wait-send-obvious1*:
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*) =
(*th-flag*(*WAIT-SEND*$_{id}$(*init-act-info caller partner*
σ(|*current-thread* := *caller*,
thread-list := *th-list*,
error-codes := *error*|))
(*IPC WAIT* (*SEND caller partner msg*))))

by (*simp add*: *WAIT-SEND*$_{id}$*-def split*: *option.split*)

lemma *act-info-buf-send-obvious0*:
(*th-flag* (*BUF-SEND*$_{id}$ (*init-act-info caller partner* σ)
(*IPC BUF* (*SEND caller partner msg*)))) =
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*)
by (*simp add*: *BUF-SEND*$_{id}$*-def*)

lemma *act-info-buf-send-obvious1*:
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*) =
(*th-flag*(*BUF-SEND*$_{id}$(*init-act-info caller partner*
σ(|*current-thread* := *caller*,
thread-list := *th-list*,
error-codes := *error*|))
(*IPC BUF* (*SEND caller partner msg*))))
by (*simp add*: *BUF-SEND*$_{id}$*-def*)

lemma *act-info-done-send-obvious0*:
(*th-flag* (*exec-action*$_{id}$ (*init-act-info caller partner* σ)
(*IPC DONE* (*SEND caller partner msg*)))) =
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*)
by *simp*

lemma *act-info-done-send-obvious1*:
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*) =
(*th-flag*(*exec-action*$_{id}$(*init-act-info caller partner*
σ(|*current-thread* := *caller*,
thread-list := *th-list*,
error-codes := *error*|))
(*IPC DONE* (*SEND caller partner msg*))))
by *simp*

lemma *act-info-prep-recv-obvious0*:
state$_{id}$.*th-flag* (*PREP-RECV*$_{id}$ (*init-act-info caller partner* σ)
(*IPC PREP* (*RECV caller partner msg*))) =
(*state*$_{id}$.*th-flag* σ)(*caller* := *None*, *partner* := *None*)
by (*simp add*: *PREP-RECV*$_{id}$*-def*)

lemma *act-info-prep-recv-obvious1*:
((*state*$_{id}$.*th-flag* σ))(*caller* := *None*, *partner* := *None*) =
((*th-flag*(*PREP-RECV*$_{id}$(*init-act-info caller partner*
σ(|*current-thread* := *caller*,
thread-list := *th-list*,
error-codes := *error*|))

$(IPC\ PREP\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add\text{: }PREP\text{-}RECV_{id}\text{-}def)$

lemma *act-info-wait-recv-obvious0*:
$(th\text{-}flag\ (WAIT\text{-}RECV_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ WAIT\ (RECV\ caller\ partner\ msg)))) =$
$(\ (th\text{-}flag\ \sigma))(caller := None,\ partner := None)$
by $(simp\ add\text{: }WAIT\text{-}RECV_{id}\text{-}def\ split\text{: }option.split)$

lemma *act-info-wait-recv-obvious1*:
$(\ (state_{id}.th\text{-}flag\ \sigma))(caller := None,\ partner := None) =$
$((th\text{-}flag(WAIT\text{-}RECV_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := th\text{-}list,$
$error\text{-}codes := error|\!))$
$(IPC\ WAIT\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add\text{: }WAIT\text{-}RECV_{id}\text{-}def\ split\text{: }option.split)$

lemma *act-info-buf-recv-obvious0*:
$(th\text{-}flag\ (BUF\text{-}RECV_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ BUF\ (RECV\ caller\ partner\ msg)))) =$
$(\ (th\text{-}flag\ \sigma))(caller := None,\ partner := None)$
by $(simp\ add\text{: }BUF\text{-}RECV_{id}\text{-}def)$

lemma *act-info-buf-recv-obvious1*:
$(\ (th\text{-}flag\ \sigma))(caller := None,\ partner := None) =$
$((th\text{-}flag(BUF\text{-}RECV_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := th\text{-}list,$
$error\text{-}codes := error|\!))$
$(IPC\ BUF\ (RECV\ caller\ partner\ msg)))))$
by $(simp\ add\text{: }BUF\text{-}RECV_{id}\text{-}def)$

lemma *act-info-done-recv-obvious0*:
$(th\text{-}flag\ (exec\text{-}action_{id}\ (init\text{-}act\text{-}info\ caller\ partner\ \sigma)$
$(IPC\ DONE\ (RECV\ caller\ partner\ msg)))) =$
$(\ (th\text{-}flag\ \sigma))(caller := None,\ partner := None)$
by *simp*

lemma *act-info-done-recv-obvious1*:
$(\ (th\text{-}flag\ \sigma))(caller := None,\ partner := None) =$
$((th\text{-}flag(exec\text{-}action_{id}(init\text{-}act\text{-}info\ caller\ partner$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := th\text{-}list,$
$error\text{-}codes := error|\!))$
$(IPC\ DONE\ (RECV\ caller\ partner\ msg)))))$

by *simp*

lemmas *atomic-action-normalizer-errors* =
PREP-RECV$_{id}$*-obvious0 PREP-RECV*$_{id}$*-obvious1 PREP-RECV*$_{id}$*-obvious2*
PREP-SEND$_{id}$*-obvious0 PREP-SEND*$_{id}$*-obvious1 PREP-SEND*$_{id}$*-obvious2*
WAIT-RECV$_{id}$*-obvious0 WAIT-RECV*$_{id}$*-obvious1 WAIT-RECV*$_{id}$*-obvious2*
WAIT-SEND$_{id}$*-obvious0 WAIT-SEND*$_{id}$*-obvious1 WAIT-SEND*$_{id}$*-obvious2*
BUF-RECV$_{id}$*-obvious0 BUF-SEND*$_{id}$*-obvious0 DONE-SEND*$_{id}$*-obvious0*
DONE-RECV$_{id}$*-obvious0*

lemmas *atomic-action-normalizer-errors-Pure* =
PREP-RECV$_{id}$*-Pure-obvious0 PREP-RECV*$_{id}$*-Pure-obvious1*
PREP-SEND$_{id}$*-Pure-obvious0 PREP-SEND*$_{id}$*-Pure-obvious1*
WAIT-RECV$_{id}$*-Pure-obvious0 WAIT-RECV*$_{id}$*-Pure-obvious1*
WAIT-SEND$_{id}$*-Pure-obvious0*
DONE-SEND$_{id}$*-Pure-obvious0*
DONE-RECV$_{id}$*-Pure-obvious0*

lemmas *atomic-action-normalizer-act-info* =
act-info-obvious0 act-info-obvious1 act-info-obvious2
act-info-prep-send-obvious0 act-info-prep-recv-obvious0
act-info-wait-send-obvious0 act-info-wait-recv-obvious0
act-info-buf-send-obvious0 act-info-buf-recv-obvious0
act-info-done-send-obvious0 act-info-done-recv-obvious0

lemmas *atomic-action-normalizer* =
prep-send-obvious prep-recv-obvious wait-send-obvious wait-recv-obvious
buf-send-obvious buf-recv-obvious

lemmas *PREP-SEND*$_{id}$*-normalizer-hyps* =
thread-eq-def
exec-action$_{id}$*-Mon-prep-fact0-def exec-action*$_{id}$*-Mon-prep-fact1-def IPC-params-c1-def*
IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def is-part-addr-th-mem-def
is-part-mem-th-def
is-part-addr-addr-def is-part-mem-def Product-Type.split-beta

lemmas *PREP-RECV*$_{id}$*-normalizer-hyps* =
thread-eq-def Product-Type.split-beta
exec-action$_{id}$*-Mon-prep-fact0-def exec-action*$_{id}$*-Mon-prep-fact1-def IPC-params-c1-def*
IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def is-part-addr-th-mem-def
is-part-mem-th-def
is-part-addr-addr-def is-part-mem-def

lemmas *WAIT-SEND*$_{id}$*-normalizer-hyps* =
thread-eq-def Product-Type.split-beta

IPC-send-comm-check-st$_{id}$-def IPC-params-c4-def IPC-buf-check-st$_{id}$-def

lemmas *WAIT-RECV$_{id}$-normalizer-hyps =*
thread-eq-def Product-Type.split-beta
IPC-recv-comm-check-st$_{id}$-def IPC-params-c4-def IPC-buf-check-st$_{id}$-def

lemmas *BUF-SEND$_{id}$-normalizer-hyps =*
thread-eq-def Product-Type.split-beta HOL.split-if HOL.split-if-asm
upd-st-res-equiv$_{id}$-def update-th-smm-equiv-def
equiv-def sym-def refl-on-def

lemmas *BUF-RECV$_{id}$-normalizer-hyps = BUF-SEND$_{id}$-normalizer-hyps*

lemmas *splitter =*
option.split errors.split
split-if list.split

lemmas *splitter-asm =*
option.split-asm errors.split-asm
split-if-asm list.split-asm

K *IPC* pre-conditions normalizer

lemmas *pre-conditions-defs =*
IPC-params-c1-def IPC-params-c2-def IPC-params-c3-def IPC-params-c4-def IPC-params-c5-def
IPC-send-comm-check-st$_{id}$-def IPC-recv-comm-check-st$_{id}$-def IPC-buf-check-st$_{id}$-def
Product-Type.split-beta is-part-addr-th-mem-def is-part-addr-addr-def

end

theory *IPC-trace-normalizer*

imports *IPC-atomic-action-normalizer*

begin

L The Core Theory for Symbolic Execution of $abort_{lift}$

L.1 mbind and ioprog fail

lemma *mbind$_{FailSave}$-ioprog-None1*:
assumes *ioprog-fail*: *ioprog a σ = None*

shows $mbind_{FailSave}\ (a\ \#\ S)\ ioprog\ \sigma = Some\ ([],\ \sigma)$
using *assms*
by(*simp add*: *Product-Type.split-beta*)

lemma $mbind_{FailSave}$*-ioprog-None2*:
assumes *exec-fail*: $mbind_{FailSave}\ (a\ \#\ S)\ ioprog\ \sigma = Some\ ([],\ \sigma)$
shows *ioprog a* $\sigma = None$
using *exec-fail*
by(*simp add*: *Product-Type.split-beta split*: *option.split-asm*)

lemma $mbind_{FailSave}$*-ioprog-None*:
$(ioprog\ a\ \sigma = None) = (mbind_{FailSave}\ (a\ \#\ S)\ ioprog\ \sigma = Some\ ([],\ \sigma))$
by (*auto simp*: $mbind_{FailSave}$*-ioprog-None1* $mbind_{FailSave}$*-ioprog-None2*)

Here is a collection of generic symbolic execution rules for for our Monad-transformer $abort_{lift}$. They make the specific semantics of aborting atomic actions explicit on the level of a side-calculus.

lemma *abort-None1*:
assumes *ioprog-fail*: *ioprog a* $\sigma = None$
shows $mbind\ (a\ \#\ S)(abort_{lift}\ ioprog)\ \sigma =$
$Some\ ([],\ \sigma)$
oops

lemma *abort-None2*:
assumes *exec-fail* : $mbind\ (a\ \#\ S)(abort_{lift}\ ioprog)\ \sigma =$
$Some([],\ \sigma)$
shows *ioprog a* $\sigma = None$
proof (*cases a*)
case (*IPC ipc-stage ipc-direction*)
assume *hyp0*: $a = IPC\ ipc\text{-}stage\ ipc\text{-}direction$
then show *?thesis*
using *assms*
proof (*cases ipc-stage*)
case *PREP*
assume *hyp1*:*ipc-stage* = *PREP*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ipc-direction*)
case (*SEND thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *SEND thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp-all add*: *Product-Type.split-beta*
split: *split-if-asm option.split-asm errors.split-asm*)
next
case (*RECV thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *RECV thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*

```
    by (simp-all add: Product-Type.split-beta
          split: split-if-asm option.split-asm errors.split-asm)
  qed
next
  case WAIT
  assume hyp1:ipc-stage = WAIT
  then show ?thesis
  using assms hyp0 hyp1
 proof (cases ipc-direction)
    case (SEND thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = SEND thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
          split: split-if-asm option.split-asm errors.split-asm)
  next
    case (RECV thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = RECV thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
          split: split-if-asm option.split-asm errors.split-asm)
  qed
next
  case BUF
  assume hyp1:ipc-stage = BUF
  then show ?thesis
  using assms hyp0 hyp1
  proof (cases ipc-direction)
    case (SEND thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = SEND thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
          split: split-if-asm option.split-asm errors.split-asm)
  next
    case (RECV thread-id1 thread-id2 adresses)
    assume hyp2: ipc-direction = RECV thread-id1 thread-id2 adresses
    then show ?thesis
    using assms hyp0 hyp1 hyp2
    by (simp-all add: Product-Type.split-beta
          split: split-if-asm option.split-asm errors.split-asm)
  qed
next
  case MAP
  assume hyp1:ipc-stage = MAP
  then show ?thesis
  using assms hyp0 hyp1
  proof (cases ipc-direction)
```

case (*SEND thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *SEND thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp-all add*: *Product-Type.split-beta*
split: *split-if-asm option.split-asm errors.split-asm*)
next
case (*RECV thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *RECV thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp-all add*: *Product-Type.split-beta*
split: *split-if-asm option.split-asm errors.split-asm*)
qed
next
case *DONE*
assume *hyp1*: *ipc-stage* = *DONE*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ipc-direction*)
case (*SEND thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *SEND thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp-all add*: *Product-Type.split-beta*
split: *split-if-asm option.split-asm errors.split-asm*)
next
case (*RECV thread-id1 thread-id2 adresses*)
assume *hyp2*: *ipc-direction* = *RECV thread-id1 thread-id2 adresses*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp-all add*: *Product-Type.split-beta*
split: *split-if-asm option.split-asm errors.split-asm*)
qed
qed
qed

lemma *abort-None′*:
assumes *not-in-err* : *caller* $\notin$ *dom* (($state_{id}$.*th-flag* σ))
and *not-done-act*: *stages* $\neq$ *DONE*
and *ioprog-fail* : *ioprog* (*IPC stages* (*SEND caller partner msg*)) σ = *None*
shows ($abort_{lift}$ *ioprog*) (*IPC stages* (*SEND caller partner msg*)) σ = *None*
using *assms*
by(*simp add*: *split*: $p4\text{-}stage_{ipc}$.*split*,*safe*, *simp-all*)

lemma *abort-None″*:
assumes *not-in-err* : $\bigwedge$*caller*. *caller* $\notin$ *dom* (($state_{id}$.*th-flag* σ))
and *not-done-act*: *stages* $\neq$ *DONE*

and *ioprog-fail* : *ioprog* (*IPC stages direction*) σ = *None*
shows (*abort*$_{lift}$ *ioprog*) (*IPC stages direction*) σ = *None*
proof (*cases stages*)
case (*PREP*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *PREP* $\Longrightarrow$
caller $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *SEND thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *PREP* $\Longrightarrow$
caller $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *RECV thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed
next
case (*WAIT*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
proof (*cases direction*)
case (*SEND thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *WAIT* $\Longrightarrow$
caller $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *SEND thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)

```
    fix caller
     show
     stages = WAIT ⟹
      caller ∉ dom ( (state_id.th-flag σ)) ⟹
      stages ≠ DONE ⟹
      ioprog (IPC stages direction) σ = None ⟹
      direction = RECV thread-id1 thread-id2 adresses ⟹
      abort_lift ioprog (IPC stages direction) σ = None
     using assms
     by simp
   qed
next
  case (BUF)
  then show abort_lift ioprog (IPC stages direction) σ = None
  using assms
    proof (cases direction)
      case (SEND thread-id1 thread-id2 adresses)
      fix caller
      show
      stages = BUF ⟹
       caller ∉ dom ( (state_id.th-flag σ)) ⟹
       stages ≠ DONE ⟹
       ioprog (IPC stages direction) σ = None ⟹
       direction = SEND thread-id1 thread-id2 adresses ⟹
       abort_lift ioprog (IPC stages direction) σ = None
      using assms
      by simp
    next
     case (RECV thread-id1 thread-id2 adresses)
     fix caller
      show
      stages = BUF ⟹
       caller ∉ dom ( (state_id.th-flag σ)) ⟹
       stages ≠ DONE ⟹
       ioprog (IPC stages direction) σ = None ⟹
       direction = RECV thread-id1 thread-id2 adresses ⟹
       abort_lift ioprog (IPC stages direction) σ = None
      using assms
      by simp
    qed
next
  case (MAP)
  then show abort_lift ioprog (IPC stages direction) σ = None
  using assms
    proof (cases direction)
      case (SEND thread-id1 thread-id2 adresses)
      fix caller
      show
      stages = MAP ⟹
```

caller $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *SEND thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
next
case (*RECV thread-id1 thread-id2 adresses*)
fix *caller*
show
stages = *MAP* $\Longrightarrow$
caller $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ)) $\Longrightarrow$
stages $\neq$ *DONE* $\Longrightarrow$
ioprog (*IPC stages direction*) σ = *None* $\Longrightarrow$
direction = *RECV thread-id1 thread-id2 adresses* $\Longrightarrow$
abort$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed
next
case (*DONE*)
then show *abort*$_{lift}$ *ioprog* (*IPC stages direction*) σ = *None*
using *assms*
by *simp*
qed

lemma *abort-None0*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *not-done-act*:*stages* $\neq$ *DONE*
and *ioprog-fail* :*ioprog* (*IPC stages* (*SEND caller partner msg*)) σ = *None*
shows (*abort*$_{lift}$ *ioprog*) (*IPC stages* (*SEND caller partner msg*)) σ =
ioprog (*IPC stages* (*SEND caller partner msg*)) σ
using *not-in-err not-done-act ioprog-fail*
by(*simp add*: *split*: *IPC-atomic-actions.p4-stage*$_{ipc}$.*split*,*safe*, *simp-all*)

lemma *abort-None1*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*state*$_{id}$.*th-flag* σ))
and *ioprog-fail*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *None*
shows *mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*)
σ =
Some ([], σ)
using *assms*
by *simp*

lemma *mbind-exec-action*$_{id}$*-Mon-None*:
mbind (*a* # *S*) *exec-action*$_{id}$*-Mon* σ $\neq$ *None*
by(*rule Monads.mbind-nofailure*)

lemma *mbind-exec-action$_{id}$-Mon-Some*:
$\exists$ *outs* σ'. *mbind* (*a* # *S*) *exec-action$_{id}$-Mon* σ = *Some* (*outs*,σ')
by(*insert mbind-exec-action$_{id}$-Mon-None*, *auto*)

lemma *mbindef-exec-action$_{id}$-Mon-None*:
mbind (*a* # *S*) *exec-action$_{id}$-Mon* $\sigma \neq$ *None*
by(*rule mbind-exec-action$_{id}$-Mon-None*)

lemma *mbindef-exec-action$_{id}$-Mon-Some*:
$\exists$ *outs* σ'. *mbind* (*a* # *S*) *exec-action$_{id}$-Mon* σ = *Some* (*outs*,σ')
by (*auto*, *rule action$_{ipc}$.induct*, *simp split*: *option.split*)

L.2 Symbolic Execution Rules on PREP stage

lemma *abort-prep-send-obvious0*:
assumes *not-in-err* : *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *abort$_{lift}$ ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-prep-send-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows *abort$_{lift}$ ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-MEM error-mem*, (*set-error-mem-preps caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-prep-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows *abort$_{lift}$ ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-preps caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-prep-send-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-sucess*:*ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows

mbind ((*IPC PREP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*# *fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*None*)
then show *?thesis*
using *assms*
by *simp*
next
case (*Some a*)
assume *hyp0*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
show *?thesis*
using *assms hyp0 hyp1*
proof (*case-tac aa*)
assume *hyp2*: *aa*= *NO-ERRORS*
show *?thesis*
using *assms hyp0 hyp1 hyp2*
by (*simp split*: *option.split*)
next
fix *error-memory*
assume *hyp3*: *aa* = *ERROR-MEM error-memory*
show *?thesis*
using *assms hyp0 hyp1 hyp3*
by *simp*
next
fix *error-IPC*
assume *hyp4*: *aa* = *ERROR-IPC error-IPC*
show *?thesis*
using *assms hyp0 hyp1 hyp4*
by *simp*
qed
qed
qed

lemma *abort-prep-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC PREP*(*SEND caller partner msg*))σ =
Some(*ERROR-MEM error-mem*,σ')
shows
mbind ((*IPC PREP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem* #
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ σ' *error-mem msg*))),

```
          snd(the(mbind S (abort_lift  ioprog)
                    (set-error-mem-preps caller partner σ σ' error-mem msg))))
  proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
  case (None)
  then show ?thesis
  using assms
  by simp
next
  case (Some a)
  assume hyp0: ioprog (IPC PREP (SEND caller partner msg)) σ = Some a
  then show ?thesis
  using assms hyp0
   proof (cases a)
     fix aa b
     assume hyp1: a = (aa, b)
     show ?thesis
     using assms hyp0 hyp1
      proof (case-tac aa)
        assume hyp2: aa = NO-ERRORS
        show ?thesis
        using assms hyp0 hyp1 hyp2
        by simp
     next
       fix error-memory
       assume hyp3: aa = ERROR-MEM error-memory
       show ?thesis
       using assms hyp0 hyp1 hyp3
       by (simp split: option.split)
      next
        fix error-IPC
        assume hyp4: aa = ERROR-IPC error-IPC
        show ?thesis
        using assms  hyp0 hyp1 hyp4
        by simp
      qed
   qed
qed


lemma abort-prep-send-obvious5:
  assumes not-in-err   : caller ∉ dom ( (th-flag σ))
   and       ioprog-succes: ioprog (IPC PREP (SEND caller partner msg)) σ =
Some(ERROR-IPC error-IPC, σ')
  shows    mbind ((IPC PREP (SEND caller partner msg))#S) (abort_lift  ioprog)
σ =
          Some(ERROR-IPC error-IPC# fst(the(mbind S (abort_lift  ioprog)
                                    (set-error-ipc-preps caller partner σ σ' error-IPC
msg))),
                                  snd(the(mbind S (abort_lift  ioprog)
```

```
                                    (set-error-ipc-preps caller partner σ σ' error-IPC
msg))))
proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
  case (None)
  assume hyp0: ioprog (IPC PREP (SEND caller partner msg)) σ = None
  then show ?thesis
  using assms hyp0
  by simp
next
  case (Some a)
  assume hyp0:ioprog (IPC PREP (SEND caller partner msg)) σ = Some a
  then show ?thesis
  using assms hyp0
    proof (cases a)
      fix aa b
      assume hyp1: a = (aa, b)
      show ?thesis
      using assms hyp0 hyp1
      proof(case-tac aa)
        assume hyp2: aa = NO-ERRORS
        show ?thesis
        using assms hyp0 hyp1 hyp2
        by simp
      next
        fix error-memory
        assume hyp3: aa = ERROR-MEM error-memory
        show ?thesis
        using assms hyp0 hyp1 hyp3
        by simp
      next
        fix error-IPCa
        assume hyp4: aa = ERROR-IPC error-IPCa
        show ?thesis
        using assms hyp0 hyp1 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                                (set-error-ipc-preps caller partner σ σ' error-IPC msg))
          case (None)
          assume hyp5:mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-preps caller partner σ σ' error-IPC msg) = None
          show ?thesis
          using assms hyp0 hyp1 hyp4 hyp5
          by simp
        next
          case (Some ab)
          assume hyp6: mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-preps caller partner σ σ' error-IPC msg) = Some
ab
          then show ?thesis
          using assms
```

by (*simp add*: *Product-Type.split-beta*)
qed
qed
qed
qed

lemma *abort-prep-send-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-prep-send-obvious7*:
assumes *in-err*: *caller* ∈ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
using *assms*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case (*None*)
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0*
by *simp*
qed
qed

lemma *abort-prep-send-obvious8*:
assumes *A*: ∀ *act* σ . *ioprog act* σ ≠ *None*
shows *mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* ∈ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*,

```
σ')
            then Some(NO-ERRORS#
                    fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller
σ σ'))),
                    snd(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller
σ σ'))))
                  else if ioprog (IPC PREP (SEND caller partner msg)) σ =
Some(ERROR-MEM error-mem, σ')
                  then  Some(ERROR-MEM error-mem#
                          fst(the(mbind S (abort_lift  ioprog)
                              (set-error-mem-preps caller partner σ σ' error-mem
msg)))
                        ,
                          snd(the(mbind S (abort_lift  ioprog)
                                (set-error-mem-preps caller partner σ σ' error-mem
msg))))
                  else if ioprog (IPC PREP (SEND caller partner msg)) σ =
Some(ERROR-IPC error-IPC, σ')
                  then  Some(ERROR-IPC error-IPC#
                          fst(the(mbind S (abort_lift  ioprog)
                              (set-error-ipc-preps caller partner σ σ' error-IPC
msg)))
                        ,
                          snd(the(mbind S (abort_lift  ioprog)
                                (set-error-ipc-preps caller partner σ σ' error-IPC
msg))))
                else if  ioprog (IPC PREP (SEND caller partner msg)) σ = None
                      then Some([], σ)
                           else id (mbind ((IPC PREP (SEND caller partner
msg))#S)(abort_lift  ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case (None)
  thus ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  thus ?thesis
  using A hyp0
  proof (cases a)
    fix aa b
    assume hyp0 : a = (aa, b)
    thus ?thesis
    using A hyp0
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      thus ?thesis
      by simp
    next
```

case (*Some ab*)
assume *hyp1*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ ab$
thus *?thesis*
using *A hyp0 hyp1*
proof (*cases ab*)
fix *ac ba*
assume *hyp2*: $ab = (ac,\ ba)$
thus *?thesis*
using *A hyp0 hyp1 hyp2*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
(*set-error-ipc-preps caller partner* $\sigma\ \sigma'$ *error-IPC msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp3*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*set-error-ipc-preps caller partner* $\sigma\ \sigma'$ *error-IPC msg*) =
Some ad
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3*
proof (*cases ad*)
fix *ae bb*
assume *hyp4*: $ad = (ae,\ bb)$
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
(*set-error-mem-preps caller partner* $\sigma\ \sigma'$ *error-mem msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
(*set-error-mem-preps caller partner* $\sigma\ \sigma'$ *error-mem msg*) =
Some af
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases af*)
fix *ag bc*
assume *hyp6*: $af = (ag,\ bc)$
thus *?thesis*
using *A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*error-tab-transfer caller* $\sigma\ \sigma'$))
case *None*
thus *?thesis*
by *simp*
next

```
                case (Some ah)
               assume hyp7:mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ σ') = Some ah
                thus ?thesis
                using A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
                proof (cases ah)
                  fix ai bd
                  assume hyp8: ah= (ai, bd)
                  thus ?thesis
                  using A hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7 hyp8
                  by simp
                qed
              qed
            qed
          qed
        qed
       qed
      qed
     qed
   qed
qed



lemma abort-prep-send-obvious8':
    mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift  ioprog) σ =
      (if caller ∈ dom ( (th-flag σ))
       then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
                 snd(the(mbind S (abort_lift  ioprog) σ)))
     else (case ioprog (IPC PREP (SEND caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
                Some(NO-ERRORS#
                    fst(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller σ
σ'))),
                     snd(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller
σ σ'))))
           | Some(ERROR-MEM error-mem, σ')⇒
                Some(ERROR-MEM error-mem#
                    fst(the(mbind S (abort_lift  ioprog)
                        (set-error-mem-preps caller partner σ σ' error-mem msg)))
                    ,
                    snd(the(mbind S (abort_lift ioprog)
                       (set-error-mem-preps caller partner σ σ' error-mem msg))))
           | Some(ERROR-IPC error-IPC, σ')⇒
                Some(ERROR-IPC error-IPC#
                    fst(the(mbind S (abort_lift  ioprog)
                    (set-error-ipc-preps caller partner σ σ' error-IPC msg)))
                    ,
                    snd(the(mbind S (abort_lift  ioprog)
```

```
                    (set-error-ipc-preps caller partner σ σ' error-IPC msg))))
              |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  thus ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  thus ?thesis
    proof −
     {have 1: caller ∈ dom ( (th-flag σ)) ⟶
            (case a of (outs, σ'') ⇒ Some (get-caller-error caller σ # outs, σ'')) =
             Some (get-caller-error caller σ # fst a, snd a)
      by (simp add: Product-Type.split-beta)
      thus ?thesis
      using hyp0 1
      proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
      { case None
        thus ?thesis
        using hyp0 1
        by simp
      next
        case (Some aa)
         assume hyp1: ioprog (IPC PREP (SEND caller partner msg)) σ = Some
aa
        thus ?thesis
        using hyp0 hyp1 1
        proof (cases aa)
          fix ab b
          assume hyp2: aa = (ab, b)
          thus ?thesis
          using hyp0 hyp1 hyp2 1
          proof (cases ab)
            case NO-ERRORS
            assume hyp3: ab = NO-ERRORS
            thus ?thesis
            using hyp0 hyp1 hyp2 hyp3 1
           proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ b))
              case None
              thus ?thesis
              by simp
            next
              case (Some ac)
               assume hyp6: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ b) = Some ac
              thus ?thesis
              using hyp0 hyp1 hyp2 hyp3 hyp6 1
```

proof (*cases a*)
fix *ad ba*
assume *hyp7*: $a = (ad, ba)$
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp7 hyp6 1*
proof (*cases ac*)
fix *ae bb*
assume *hyp8*: $ac = (ae, bb)$
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp7 hyp6 hyp8 1*
by *simp*
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*: *ab = ERROR-MEM error-memory*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *b*)
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *b = Some ac*
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp6 1*
proof (*cases a*)
fix *ad ba*
assume *hyp7*: $a = (ad, ba)$
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 1*
proof (*cases ac*)
fix *ae bb*
assume *hyp8*: $ac = (ae, bb)$
thus *?thesis*
using *hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *b error-memory msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *b error-memory msg*) =
Some af

```
          thus ?thesis
          using hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 hyp9 1
          proof (cases af)
            fix ag bc
            assume hyp10:  af = (ag, bc)
            thus ?thesis
            using hyp0 hyp1 hyp2 hyp4 hyp7 hyp6 hyp8 hyp9 hyp10 1
            by simp
          qed
        qed
      qed
    qed
  qed
next
  case (ERROR-IPC error-IPC)
  assume hyp5:  ab = ERROR-IPC error-IPC
  thus ?thesis
  using hyp0 hyp1 hyp2 hyp5 1
  proof (cases mbind_FailSave S (abort_lift ioprog) b)
    case None
    thus ?thesis
    by simp
  next
    case (Some ac)
    assume hyp6: mbind_FailSave S (abort_lift ioprog) b = Some ac
    thus ?thesis
    using hyp0 hyp1 hyp2 hyp5 hyp6 1
    proof (cases a)
      fix ad ba
      assume hyp7: a = (ad, ba)
      thus ?thesis
      using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 1
      proof (cases ac)
        fix ae bb
        assume hyp8: ac = (ae, bb)
        thus ?thesis
        using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 1
        proof (cases mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-preps caller partner σ b error-IPC msg))
          case None
          thus ?thesis
          by simp
        next
          case (Some af)
          assume hyp9: mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-preps caller partner σ b error-IPC msg) =
                     Some af
          thus ?thesis
          using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 hyp9 1
```

```
                proof (cases af)
                  fix ag bc
                  assume hyp10:  af = (ag, bc)
                  thus ?thesis
                  using hyp0 hyp1 hyp2 hyp5 hyp7 hyp6 hyp8 hyp9 hyp10 1
                  by simp
                qed
              qed
            qed
          qed
        qed
      qed
    }qed
  }qed
qed



lemma abort-prep-send-obvious9:
  fst(the(mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift  ioprog)
σ)) =
      (if caller ∈ dom ( (th-flag σ))
         then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

    else (case ioprog (IPC PREP (SEND caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              NO-ERRORS#
            fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ σ')))
        | Some(ERROR-MEM error-mem, σ')⇒
            ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)
                    (set-error-mem-preps caller partner σ σ' error-mem
msg)))
        | Some(ERROR-IPC error-IPC, σ')⇒
            ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                      (set-error-ipc-preps caller partner σ σ' error-IPC
msg)))
        |  None ⇒ []))
proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
  case None
  thus ?thesis
  using assms
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    assume hyp0:  ioprog (IPC PREP (SEND caller partner msg)) σ = None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = None
    thus ?thesis
    using assms hyp0 hyp1
    by simp
```

```
  next
    case (Some a)
    assume hyp0:  ioprog (IPC PREP (SEND caller partner msg)) σ = None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some a
    thus  ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      thus ?thesis
      using assms hyp0 hyp1  hyp2
      by simp
    qed
  qed
next
  case (Some a)
  assume hyp0: ioprog (IPC PREP (SEND caller partner msg)) σ = Some a
  thus ?thesis
  using hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = None
    thus ?thesis
    using assms hyp1 hyp0
    by simp
  next
    case (Some aa)
    assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some aa
    thus ?thesis
    using hyp0 hyp2 assms
    proof −
      have 1: (caller ∈ dom ( (th-flag σ)) ⟶
                fst (the (case aa of (outs, σ'') ⇒ Some (get-caller-error caller σ #
outs, σ'')))) =
               get-caller-error caller σ # fst aa)
               proof (cases aa)
                 fix a b
                 assume hyp3: aa = (a, b)
                 thus ?thesis
                 by simp
               qed
      thus ?thesis
      using 1 assms hyp0 hyp2
      proof (cases a)
        fix ab b
        assume hyp3:a = (ab, b)
        thus ?thesis
        using hyp3  1 assms hyp0 hyp2
        proof (cases ab)
```

```
        case (NO-ERRORS)
        thus ?thesis
        using hyp3 1 assms hyp0 hyp2
        proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ b))
          case None
          thus ?thesis
          by simp
        next
          case (Some ac)
          assume hyp4:ab = NO-ERRORS
            assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ b) = Some ac
          thus ?thesis
          using hyp3 hyp4 hyp5 1assms hyp0 hyp2
          proof (cases ac)
            fix ad ba
            assume hyp6: ac = (ad, ba)
            thus ?thesis
            using hyp3 hyp4 hyp5 1 assms hyp0 hyp2
            by simp
          qed
        qed
      next
        case (ERROR-MEM error-memory)
        thus ?thesis
        using hyp3 1 assms hyp0 hyp2
        proof (cases mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-preps caller partner σ b error-memory msg))
          case None
          thus ?thesis
          by simp
        next
          case (Some ac)
          assume hyp7: ab = ERROR-MEM error-memory
          assume hyp8: mbind_FailSave S (abort_lift ioprog)
                      (set-error-mem-preps caller partner σ b error-memory msg)
= Some ac
          thus ?thesis
          using hyp3 hyp8 hyp7 1 assms hyp0 hyp2
          proof (cases ac)
            fix ad ba
            assume hyp6: ac = (ad, ba)
            thus ?thesis
            using hyp3 hyp8 hyp7 1 assms hyp0 hyp2
            by simp
          qed
        qed
      next
```

```
          case (ERROR-IPC error-IPC)
          thus ?thesis
          using hyp3 1 assms hyp0 hyp2
          proof (cases mbind_FailSave S (abort_lift ioprog)
                      (set-error-ipc-preps caller partner σ b error-IPC msg))
            case None
            thus ?thesis
            by simp
          next
            case (Some ac)
            assume hyp9: mbind_FailSave S (abort_lift ioprog)
                      (set-error-ipc-preps caller partner σ b error-IPC msg) = Some
ac
            assume hyp10: ab = ERROR-IPC error-IPC
            thus ?thesis
            using assms hyp9 hyp10 hyp3 1  hyp0 hyp2
           proof (cases ac)
             fix ad ba
             assume hyp6: ac = (ad, ba)
             thus ?thesis
             using hyp3 hyp9 hyp10 1 assms hyp0 hyp2
             by simp
           qed
          qed
        qed
      qed
    qed
  qed
qed



lemma abort-prep-recv-obvious0:
  assumes not-in-err  : caller ∉ dom ( (th-flag σ))
   and       ioprog-succes:ioprog (IPC PREP (RECV caller partner msg)) σ =
Some(NO-ERRORS, σ')
 shows  abort_lift  ioprog (IPC PREP (RECV caller partner msg)) σ = Some(NO-ERRORS,
(error-tab-transfer caller σ σ'))
  using assms
  by simp

lemma abort-prep-recv-obvious1:
  assumes not-in-err :caller ∉ dom ( (th-flag σ))
  and ioprog-success :ioprog (IPC PREP (RECV caller partner msg)) σ =
                    Some(ERROR-MEM error-mem, σ')
  shows    abort_lift  ioprog (IPC PREP (RECV caller partner msg)) σ =
          Some (ERROR-MEM error-mem, (set-error-mem-prepr caller partner σ
σ' error-mem msg))
  using assms
```

by *simp*

lemma *abort-prep-recv-obvious2*:
assumes *not-in-err* : *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-prep-recv-obvious3*:
assumes *not-in-err* : *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*# *fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show*?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-prep-recv-obvious4*:
assumes *not-in-err* : *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem*

msg))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases* *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some aa*)
assume *hyp1*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*) = *Some aa*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases aa*)
fix *ab b*
assume *hyp2*: *aa* = (*ab*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
qed

lemma *abort-prep-recv-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next

case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
(*set-error-ipc-prepr caller partner* $\sigma\ \sigma'$ *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some aa*)
assume *hyp1*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
(*set-error-ipc-prepr caller partner* $\sigma\ \sigma'$ *error-IPC msg*) =
Some aa
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases aa*)
fix *ab b*
assume *hyp2*: *aa* = (*ab*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
qed

lemma *abort-prep-recv-obvious6*:
assumes *in-err*: *caller* ∈ *dom* ((*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC PREP* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *in-err*
by *simp*

lemma *abort-prep-recv-obvious7*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC PREP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*

```
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    by simp
  qed
qed

lemma abort-prep-recv-obvious8:
     mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift  ioprog) σ =
        (if caller ∈ dom ( (th-flag σ))
        then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
                  snd(the(mbind S (abort_lift  ioprog) σ)))

      else if  ioprog (IPC PREP (RECV caller partner msg)) σ = Some(NO-ERRORS,
σ')
             then Some(NO-ERRORS#
                       fst(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller σ
σ'))),
                        snd(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller
σ σ'))))
          else if ioprog (IPC PREP (RECV caller partner msg)) σ = Some(ERROR-MEM
error-mem, σ')
                 then  Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift
ioprog)
                                    (set-error-mem-prepr caller partner σ σ' error-mem
msg)))
                      ,
                       snd(the(mbind S (abort_lift  ioprog)
                                    (set-error-mem-prepr caller partner σ σ' error-mem
msg))))
                       else if ioprog (IPC PREP (RECV caller partner msg)) σ =
Some(ERROR-IPC error-IPC, σ')
                   then  Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift
ioprog)
                                 (set-error-ipc-prepr caller partner σ σ' error-IPC msg)))
                      ,
                       snd(the(mbind S (abort_lift  ioprog)
                            (set-error-ipc-prepr caller partner σ σ' error-IPC msg))))
                  else if   ioprog (IPC PREP (RECV caller partner msg)) σ = None
                       then Some([], σ)
                            else id (mbind ((IPC PREP (RECV caller partner
msg))#S)(abort_lift  ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ )
  case None
  then show ?thesis
  by simp
next
```

case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ σ' *error-IPC msg*) = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ σ' *error-mem msg*) =*Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*:*ad* = (*ae*,*bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)

```
            assume hyp6:mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ σ') =  Some af
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
          proof (cases af)
            fix ag bc
            assume hyp7:af = (ag,bc)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
            by simp
          qed
        qed
      qed
    qed
  qed
 qed
qed


lemma abort-prep-recv-obvious8':
    mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift  ioprog) σ =
      (if caller ∈ dom ( (th-flag σ))
       then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
            snd(the(mbind S (abort_lift  ioprog) σ)))
     else (case ioprog (IPC PREP (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              Some(NO-ERRORS#
                 fst(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller σ
σ'))),
                  snd(the(mbind S (abort_lift  ioprog)  (error-tab-transfer caller
σ σ'))))
          | Some(ERROR-MEM error-mem, σ')⇒
            Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)

                                        (set-error-mem-prepr caller partner σ σ'
error-mem msg)))
                  ,
                  snd(the(mbind S (abort_lift  ioprog)
                     (set-error-mem-prepr caller partner σ σ' error-mem msg))))
          | Some(ERROR-IPC error-IPC, σ')⇒
             Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                              (set-error-ipc-prepr caller partner σ σ' error-IPC
msg)))
                  ,
                  snd(the(mbind S (abort_lift  ioprog)
                       (set-error-ipc-prepr caller partner σ σ' error-IPC msg))))
          |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
```

```
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1:a= (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by simp
    next
      case (Some ab)
      assume hyp2:ioprog (IPC PREP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac,ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4:ac  = NO-ERRORS
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
              assume hyp7: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ ba)= Some ad
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7
            proof (cases ad)
              fix ae bb
              assume hyp8:ad =  (ae, bb)
              then show ?thesis
              using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8
```

by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*: *ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*:*ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*: *ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*) =
Some ad
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*:*ad* = (*ae*,*bb*)
then show *?thesis*

```
            using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
            by simp
          qed
        qed
      qed
    qed
  qed
qed
```

```
lemma abort-prep-recv-obvious9:
  fst(the(mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift  ioprog)
σ)) =
      (if caller ∈ dom ( (th-flag σ))
         then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

     else (case ioprog (IPC PREP (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
                  NO-ERRORS#
                fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ σ')))
           | Some(ERROR-MEM error-mem, σ')⇒
               ERROR-MEM error-mem#
               fst(the(mbind S (abort_lift  ioprog)
               (set-error-mem-prepr caller partner σ σ' error-mem msg)))
           | Some(ERROR-IPC error-IPC, σ')⇒
               ERROR-IPC error-IPC#
               fst(the(mbind S (abort_lift  ioprog)
               (set-error-ipc-prepr caller partner σ σ' error-IPC msg)))
           |  None ⇒ []))
proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
  case None
  thus ?thesis
  using assms
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    assume hyp0:  ioprog (IPC PREP (RECV caller partner msg)) σ = None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = None
    thus ?thesis
    using assms hyp0 hyp1
    by simp
  next
    case (Some a)
    assume hyp0:  ioprog (IPC PREP (RECV caller partner msg)) σ = None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some a
    thus  ?thesis
    using assms hyp0 hyp1
```

```
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      thus ?thesis
      using assms hyp0 hyp1  hyp2
      by simp
    qed
  qed
next
  case (Some a)
  assume hyp0: ioprog (IPC PREP (RECV caller partner msg)) σ = Some a
  thus ?thesis
  using hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = None
    thus ?thesis
    using assms hyp1 hyp0
    by simp
  next
    case (Some aa)
    assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some aa
    thus ?thesis
    using hyp0 hyp2 assms
    proof −
      have 1: (caller ∈ dom ( (th-flag σ)) ⟶
               fst (the (case aa of (outs, σ'') ⇒ Some (get-caller-error caller σ #
outs, σ'')))) =
              get-caller-error caller σ # fst aa)
              proof (cases aa)
                fix a b
                assume hyp3: aa = (a, b)
                thus ?thesis
                by simp
              qed
      thus ?thesis
      using 1 assms hyp0 hyp2
      proof (cases a)
        fix ab b
        assume hyp3:a = (ab, b)
        thus ?thesis
        using hyp3  1 assms hyp0 hyp2
        proof (cases ab)
          case (NO-ERRORS)
          thus ?thesis
          using hyp3 1 assms hyp0 hyp2
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ b))
            case None
```

thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp4*:*ab* = *NO-ERRORS*
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *b*) = *Some ac*
thus *?thesis*
using *hyp3 hyp4 hyp5 1assms hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: *ac* = (*ad*, *ba*)
thus *?thesis*
using *hyp3 hyp4 hyp5 1 assms hyp0 hyp2*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*set-error-mem-prepr caller partner* σ *b error-memory msg*))
case *None*
thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp7*: *ab* = *ERROR-MEM error-memory*
assume *hyp8*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*set-error-mem-prepr caller partner* σ *b error-memory msg*) = *Some ac*
thus *?thesis*
using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: *ac* = (*ad*, *ba*)
thus *?thesis*
using *hyp3 hyp8 hyp7 1 assms hyp0 hyp2*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
thus *?thesis*
using *hyp3 1 assms hyp0 hyp2*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*set-error-ipc-prepr caller partner* σ *b error-IPC msg*))
case *None*

thus *?thesis*
by *simp*
next
case (*Some ac*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *b error-IPC msg*) = *Some*
ac
assume *hyp10*: *ab* = *ERROR-IPC error-IPC*
thus *?thesis*
using *assms hyp9 hyp10 hyp3 1 hyp0 hyp2*
proof (*cases ac*)
fix *ad ba*
assume *hyp6*: *ac* = (*ad*, *ba*)
thus *?thesis*
using *hyp3 hyp9 hyp10 1 assms hyp0 hyp2*
by *simp*
qed
qed
qed
qed
qed
qed

L.3 Symbolic Execution rules on WAIT stage

lemma *abort-wait-send-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*,
(*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-wait-send-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-succes*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-waits caller partner* σ
σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-wait-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')

shows $abort_{lift}\ \ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ (set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'$
$error\text{-}IPC\ msg))$
using *assms*
by *simp*

lemma *abort-wait-send-obvious3*:
assumes *not-in-err*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-sucess*:$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(NO\text{-}ERRORS,\ \sigma')$
shows $mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)$
$\sigma =$
$Some(NO\text{-}ERRORS\#\ fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer$
$caller\ \sigma\ \sigma')))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller$
$\sigma\ \sigma'))))$
using *assms*
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma$
$\sigma') = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*:$a = (aa,b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious4*:
assumes *not-in-err*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma')$
shows $mbind\ ((IPC\ WAIT\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)$
$\sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem$
$msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$

case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a*= (*aa*,*b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious5*:
assumes *not-in-err*: *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*)
σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ σ' *error-IPC*
msg))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*) =
Some a
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed

qed

lemma *abort-wait-send-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-wait-send-obvious7*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-send-obvious8*:
mbind ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* ∈ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ′)
then Some(*NO-ERRORS*#
fst(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ′))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ′))))
else if ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ′)
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$

```
ioprog)
                                (set-error-mem-waits caller partner σ σ' error-mem
msg)))
                        ,
                         snd(the(mbind S (abort_lift  ioprog)
                                  (set-error-mem-waits caller partner σ σ' error-mem
msg))))
                         else if ioprog (IPC WAIT (SEND caller partner msg)) σ =
Some(ERROR-IPC error-IPC, σ')
                    then  Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift
ioprog)
                                  (set-error-ipc-waits caller partner σ σ' error-IPC
msg)))
                        ,
                         snd(the(mbind S (abort_lift  ioprog)
                              (set-error-ipc-waits caller partner σ σ' error-IPC msg))))
                  else if  ioprog (IPC WAIT (SEND caller partner msg)) σ = None
                       then Some([], σ)
                            else id (mbind ((IPC WAIT (SEND caller partner
msg))#S)(abort_lift  ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa,b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-waits caller partner σ σ' error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case  (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-waits caller partner σ σ' error-IPC msg) =
Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
```

then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ σ' *error-mem msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*: *af* = (*ag*, *bc*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious8'*:
mbind ((*IPC WAIT* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =

```
      (if caller ∈ dom ( (th-flag σ))
      then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
            snd(the(mbind S (abort_lift  ioprog) σ)))

    else (case ioprog (IPC WAIT (SEND caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              Some(NO-ERRORS#
            fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ σ'))),
               snd(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ
σ'))))
          | Some(ERROR-MEM error-mem, σ')⇒
            Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)

                                        (set-error-mem-waits caller partner σ σ'
error-mem msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                              (set-error-mem-waits caller partner σ σ' error-mem
msg))))
          | Some(ERROR-IPC error-IPC, σ')⇒
             Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                                 (set-error-ipc-waits caller partner σ σ' error-IPC
msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                          (set-error-ipc-waits caller partner σ σ' error-IPC msg))))
          |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1:a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases ioprog (IPC WAIT (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by simp
    next
      case (Some ab)
      assume hyp2: ioprog (IPC WAIT (SEND caller partner msg)) σ = Some ab
```

then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)

fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious9*:
fst(*the*(*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* ((*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ))

else (*case ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
NO-ERRORS#

$fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
| $Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma') \Rightarrow$
$ERROR\text{-}MEM\ error\text{-}mem \# fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
| $Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \Rightarrow$
$ERROR\text{-}IPC\ error\text{-}IPC \# fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waits\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
| $None \Rightarrow [])$
by (*simp split*: *option.split errors.split*, *auto*)

lemma *abort-wait-recv-obvious0*:
assumes *not-in-err* :$caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ \sigma')$
shows $abort_{lift}\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
using *assms*
by *simp*

lemma *abort-wait-recv-obvious1*:
assumes *not-in-err* :$caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*: $ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma')$
shows $abort_{lift}\ \ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM\ error\text{-}mem,\ (set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))$
using *assms*
by *simp*

lemma *abort-wait-recv-obvious2*:
assumes *not-in-err* :$caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')$
shows $abort_{lift}\ \ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ (set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
using *assms*
by *simp*

lemma *abort-wait-recv-obvious3*:
assumes *not-in-err*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS,\ \sigma')$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$

Some(*NO-ERRORS*# *fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious4*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ' *error-mem msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*

using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious5*:
assumes *not-in-err*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma')$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa,\ b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious6*:
assumes *in-err*:$caller \in dom\ (\ (th\text{-}flag\ \sigma))$
shows $abort_{lift}\ \ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma,\ \sigma)$
using *assms*
by *simp*

lemma *abort-wait-recv-obvious7*:
assumes *in-err*:$caller \in dom\ (\ (th\text{-}flag\ \sigma))$
shows $mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)\ \sigma =$
$Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$

case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-wait-recv-obvious8*:
$mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma =$
$(if\ caller \in dom\ (\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$

$else\ if\ \ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
$then\ Some(NO\text{-}ERRORS\#$
$fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
$then\ \ Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$else\ if\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
$then\ \ Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))))$
$else\ if\ \ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None$

```
                then Some([], σ)
                     else id (mbind ((IPC WAIT (RECV caller partner
msg))#S)(abort_lift  ioprog) σ)
  )
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa,b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog)
                        (set-error-ipc-waitr caller partner σ σ' error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog)
                        (set-error-ipc-waitr caller partner σ σ' error-IPC msg) =
Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-waitr caller partner σ σ' error-mem msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp4:mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-waitr caller partner σ σ' error-mem msg) = Some
ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases ad)
            fix ae bb
```

```
          assume hyp5: ad = (ae, bb)
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
         proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ σ'))
           case None
           then show ?thesis
           by simp
          next
           case (Some af)
            assume hyp6: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ σ') = Some af
           then show ?thesis
           using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
           proof (cases af)
            fix ag bc
            assume hyp7: af = (ag, bc)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
            by simp
           qed
          qed
         qed
        qed
       qed
      qed
    qed
  qed
qed



lemma abort-wait-recv-obvious8':
    mbind ((IPC WAIT (RECV caller partner msg))#S)(abort_lift  ioprog) σ =
      (if caller ∈ dom ( (th-flag σ))
      then Some(get-caller-error caller σ#
              fst(the(mbind S (abort_lift  ioprog) σ)),
              snd(the(mbind S (abort_lift  ioprog) σ)))
     else (case ioprog (IPC WAIT (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              Some(NO-ERRORS#
                  fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ
σ'))),
                  snd(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ
σ'))))
          | Some(ERROR-MEM error-mem, σ')⇒
            Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift ioprog)

                                (set-error-mem-waitr caller partner σ σ' error-mem
msg)))
```

```
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                              (set-error-mem-waitr caller partner σ σ' error-mem
msg))))
          | Some(ERROR-IPC error-IPC, σ')⇒
            Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                              (set-error-ipc-waitr caller partner σ σ' error-IPC
msg)))
                    ,
                    snd(the(mbind S (abort_lift  ioprog)
                              (set-error-ipc-waitr caller partner σ σ' error-IPC
msg))))
          |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1:a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases ioprog (IPC WAIT (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by simp
    next
      case (Some ab)
      assume hyp2: ioprog (IPC WAIT (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
```

case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))

```
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp11: mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-waitr caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11
          proof (cases ad)
            fix ae bb
            assume hyp12: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
            by simp
          qed
        qed
      qed
    qed
  qed
qed



lemma abort-wait-recv-obvious9:
   fst(the(mbind ((IPC WAIT (RECV caller partner msg))#S)(abort_lift  ioprog)
σ)) =
       (if caller ∈ dom ( (th-flag σ))
          then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

      else (case ioprog (IPC WAIT (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
                 NO-ERRORS#
               fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller σ σ')))
            | Some(ERROR-MEM error-mem, σ')⇒
                ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)
                          (set-error-mem-waitr caller partner σ σ' error-mem
msg)))

            | Some(ERROR-IPC error-IPC, σ')⇒
                ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog) (
                             set-error-ipc-waitr caller partner σ σ' error-IPC
msg)))

            |  None ⇒ []))
  by (simp split: option.split errors.split,auto)
```

L.4 Symbolic Execution rules on BUF stage

lemma *abort-buf-send-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-buf-send-obvious1*:
assumes *not-in-err* : *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some* (*ERROR-MEM error-mem*, (*set-error-mem-bufs caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-buf-send-obvious2*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-buf-send-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*

proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious4*:
assumes *not-in-err*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-success*:$ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
shows $mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)$
$\sigma =$
$Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem$
$msg))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-send-obvious5*:
assumes *not-in-err* :$caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
and *ioprog-succes* : $ioprog\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
shows $mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ ioprog)$
$\sigma =$
$Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}bufs\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC$
$msg))),$

```
                snd(the(mbind S (abort_lift  ioprog)
                      (set-error-ipc-bufs caller partner σ σ' error-IPC msg))))
proof (cases mbind_FailSave S (abort_lift ioprog)
          (set-error-ipc-dones caller partner σ σ' error-IPC msg))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)
            (set-error-ipc-dones caller partner σ σ' error-IPC msg) = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    by simp
  qed
qed

lemma abort-buf-send-obvious6:
  assumes in-err:caller ∈ dom ( (th-flag σ))
  shows   abort_lift  ioprog (IPC BUF (SEND caller partner msg)) σ =
          Some(get-caller-error caller σ, σ)
  using assms
  by simp

lemma abort-buf-send-obvious7:
  assumes in-err: caller ∈ dom ( (th-flag σ))
  shows   mbind ((IPC BUF (SEND caller partner msg))#S) (abort_lift  ioprog)
σ =
          Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
                                    snd(the(mbind S (abort_lift  ioprog) σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0:mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
```

by *simp*
qed
qed

lemma *abort-buf-send-obvious8*:
assumes *A*: ∀ *act σ* . *ioprog act σ* ≠ *None*
shows *mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) *σ* =
(*if caller* ∈ *dom* ((*th-flag σ*))
then Some(*get-caller-error caller σ*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) *σ*)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) *σ*)))

else if ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ'*)
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller σ σ'*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller σ σ'*))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ'*)
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner σ σ' error-mem msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner σ σ' error-mem msg*))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-IPC error-IPC*, *σ'*)
then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner σ σ' error-IPC msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner σ σ' error-IPC msg*))))
else if ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *None*
then Some([], *σ*)
else id (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) *σ*))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ* = *Some a*
then show *?thesis*
using *assms hyp0*

```
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa,b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog)
                              (set-error-ipc-bufs caller partner σ σ' error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog)
                              (set-error-ipc-bufs caller partner σ σ' error-IPC msg) =
Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-bufs caller partner σ σ' error-mem msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp4:mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-bufs caller partner σ σ' error-mem msg) = Some
ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases ad)
            fix ae bb
            assume hyp5: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
           proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ σ'))
              case None
              then show ?thesis
              by simp
            next
              case (Some af)
               assume hyp6: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ σ') = Some af
              then show ?thesis
```

```
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            proof (cases af)
              fix ag bc
              assume hyp7: af = (ag, bc)
              then show ?thesis
              using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
              by simp
            qed
          qed
        qed
      qed
    qed
  qed
qed



lemma abort-buf-send-obvious8':
     mbind ((IPC BUF (SEND caller partner msg))#S)(abort_lift  ioprog) σ =
       (if caller ∈ dom ( (th-flag σ))
       then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ)),
                 snd(the(mbind S (abort_lift  ioprog) σ)))

      else (case ioprog (IPC BUF (SEND caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
                          Some(NO-ERRORS#fst(the(mbind S (abort_lift  ioprog)
(error-tab-transfer caller σ σ'))),
                        snd(the(mbind S (abort_lift  ioprog) (error-tab-transfer caller
σ σ'))))
            | Some(ERROR-MEM error-mem, σ')⇒
              Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)

                                  (set-error-mem-bufs caller partner σ σ' error-mem
msg)))
                        ,
                         snd(the(mbind S (abort_lift  ioprog)
                             (set-error-mem-bufs caller partner σ σ' error-mem msg))))
            | Some(ERROR-IPC error-IPC, σ')⇒
              Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                             (set-error-ipc-bufs caller partner σ σ' error-IPC msg)))
                        ,
                         snd(the(mbind S (abort_lift  ioprog)
                               (set-error-ipc-bufs caller partner σ σ' error-IPC msg))))
            |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
```

next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed

next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner σ ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner σ ba error-memory msg*) =
Some ad
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner σ ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner σ ba error-IPC msg*) = *Some*
ad
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12*
by *simp*
qed

qed
qed
qed
qed
qed
qed

lemma *abort-buf-send-obvious9*:
fst(*the*(*mbind* (*IPC BUF* (*SEND caller partner msg*)#*S*)(*abort*$_{lift}$ *ioprog*) σ))
=
(*if caller* $\in$ *dom* ((*th-flag* σ))
then get-caller-error caller σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ))

else (*case ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
NO-ERRORS#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ')))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
ERROR-MEM error-mem#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ σ' *error-mem msg*)))

| *Some*(*ERROR-IPC error-IPC*, σ')⇒
ERROR-IPC error-IPC#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*)))

| *None* ⇒ []))
by (*simp split*: *option.split errors.split*,*auto*)

lemma *abort-buf-recv-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*NO-ERRORS*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-buf-recv-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*ERROR-MEM error-mem*, σ')
shows *abort*$_{lift}$ *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some (*ERROR-MEM error-mem*, (*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))
using *assms*

by *simp*

lemma *abort-buf-recv-obvious2*:
assumes *not-in-err*: *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-succes*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-buf-recv-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success* : *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious4*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem*

msg))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))))
using *assms*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*)
σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC*
msg))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)

then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-buf-recv-obvious7*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*)
σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-buf-recv-obvious8*:
mbind ((*IPC BUF* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
(*if caller* ∈ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*,
σ′)
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer*
caller σ σ′))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer*

```
caller σ σ'))))
        else if ioprog (IPC BUF (RECV caller partner msg)) σ = Some(ERROR-MEM
error-mem, σ')
               then  Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift
ioprog)
                                    (set-error-mem-bufr caller partner σ σ' error-mem
msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                         (set-error-mem-bufr caller partner σ σ' error-mem msg))))
                      else if ioprog (IPC BUF (RECV caller partner msg)) σ =
Some(ERROR-IPC error-IPC, σ')
                then  Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift
ioprog)
                                    (set-error-ipc-bufr caller partner σ σ' error-IPC
msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                          (set-error-ipc-bufr caller partner σ σ' error-IPC msg))))
             else if  ioprog (IPC BUF (RECV caller partner msg)) σ = None
                 then Some([], σ)
            else id (mbind ((IPC BUF (RECV caller partner msg))#S)(abort_lift
ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa,b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-bufr caller partner σ σ' error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case  (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-bufr caller partner σ σ' error-IPC msg) =
Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
```

```
    proof (cases ab)
      fix ac ba
      assume hyp3: ab = (ac,ba)
      then show ?thesis
      using assms hyp0 hyp1 hyp2 hyp3
      proof (cases mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-bufr caller partner σ σ' error-mem msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp4:mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-bufr caller partner σ σ' error-mem msg) = Some
ad
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases ad)
          fix ae bb
          assume hyp5: ad = (ae, bb)
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
         proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ σ'))
            case None
            then show ?thesis
            by simp
          next
            case (Some af)
             assume hyp6: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ σ') = Some af
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            proof (cases af)
              fix ag bc
              assume hyp7: af = (ag, bc)
              then show ?thesis
              using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
              by simp
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

lemma *abort-buf-recv-obvious8'*:
mbind ((*IPC BUF* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
else (*case ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ σ' *error-mem msg*))))
| *Some*(*ERROR-IPC error-IPC*, σ')⇒
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*))))
| *None* ⇒ *Some*([], σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next

case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*

```
          using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9
          proof (cases ad)
            fix ae bb
            assume hyp10: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10
            by simp
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp6:ac = ERROR-IPC error-IPC
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3 hyp6
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-bufr caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp11: mbind_FailSave S (abort_lift ioprog)
                      (set-error-ipc-bufr caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11
          proof (cases ad)
            fix ae bb
            assume hyp12: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
            by simp
          qed
        qed
      qed
    qed
  qed
qed



lemma abort-buf-recv-obvious9:
   fst(the(mbind ((IPC BUF (RECV caller partner msg))#S)(abort_lift  ioprog)
σ)) =
       (if caller ∈ dom ( (th-flag σ))
          then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

     else (case ioprog (IPC BUF (RECV caller partner msg)) σ of Some(NO-ERRORS,
```

σ')⇒
NO-ERRORS#fst(the(mbind S ($abort_{lift}$ ioprog) (error-tab-transfer caller σ σ')))
| Some(ERROR-MEM error-mem, σ')⇒
ERROR-MEM error-mem#fst(the(mbind S ($abort_{lift}$ ioprog)
(set-error-mem-bufr caller partner σ σ' error-mem msg)))
| Some(ERROR-IPC error-IPC, σ')⇒
ERROR-IPC error-IPC#fst(the(mbind S ($abort_{lift}$ ioprog)
(set-error-ipc-bufr caller partner σ σ' error-IPC msg)))
| None ⇒ []))
by(*simp split*: *option.split errors.split*,*auto*)

L.5 Symbolic Execution Rules on MAP stage

lemma *abort-map-send-obvious0*:
assumes *not-in-err* :*caller* ∉ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-map-send-obvious1*:
assumes *not-in-err* : *caller* ∉ *dom* ((*th-flag* σ))
and *ioprog-success*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-MEM error-mem*, (*set-error-mem-maps caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-map-send-obvious2*:
assumes *not-in-err* :*caller* ∉ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-map-send-obvious3*:
assumes *not-in-err* :*caller* ∉ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')

shows *mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:*mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious4*:
assumes *not-in-err*: *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ σ' *error-mem msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ σ' *error-mem msg*)= *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)

then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-succes* : *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-dones caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-dones caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious6*:
assumes *in-err*:*caller* $\in$ *dom* ((*th-flag* σ))
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-map-send-obvious7*:
assumes *in-err*: *caller* $\in$ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) σ)),

$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma$)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa,\ b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-send-obvious8*:
assumes *A*: $\forall\ act\ \sigma\ .\ ioprog\ act\ \sigma \neq None$
shows $mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ \ ioprog)\ \sigma =$
$(if\ caller \in dom\ (\ (th\text{-}flag\ \sigma))$
$then\ Some(get\text{-}caller\text{-}error\ caller\ \sigma\#fst(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ \sigma)))$

$else\ if\ \ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma')$
$then\ \ Some(NO\text{-}ERRORS\#fst(the(mbind\ \ S\ \ (abort_{lift}\ \ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))),$
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))))$
$else\ if\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')$
$then\ \ Some(ERROR\text{-}MEM\ error\text{-}mem\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))$
,
$snd(the(mbind\ S\ (abort_{lift}\ \ ioprog)$
$(set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))))$
$else\ if\ ioprog\ (IPC\ \ MAP\ \ (SEND\ \ caller\ \ partner\ \ msg))\ \ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')$
$then\ \ Some(ERROR\text{-}IPC\ error\text{-}IPC\#fst(the(mbind\ S\ (abort_{lift}\ ioprog)$
$(set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))$

```
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                          (set-error-ipc-maps caller partner σ σ' error-IPC msg))))
                else if  ioprog (IPC MAP (SEND caller partner msg)) σ = None
                    then Some([], σ)
                                else id (mbind ((IPC MAP (SEND caller partner
msg))#S)(abort_lift  ioprog) σ))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa,b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-maps caller partner σ σ' error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case  (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog)
                          (set-error-ipc-maps caller partner σ σ' error-IPC msg) =
Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-maps caller partner σ σ' error-mem msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp4:mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-maps caller partner σ σ' error-mem msg) = Some
ad
          then show ?thesis
```

using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases af*)
fix *ag bc*
assume *hyp7*: *af* = (*ag*, *bc*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by *simp*
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-send-obvious8'*:
mbind ((*IPC MAP* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
(*if caller* ∈ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))

else (*case ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ *of Some*(*NO-ERRORS*, σ')⇒
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
| *Some*(*ERROR-MEM error-mem*, σ')⇒
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)

```
                                        (set-error-mem-maps caller partner σ σ'
error-mem msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                                (set-error-mem-maps caller partner σ σ' error-mem
msg))))
          | Some(ERROR-IPC error-IPC, σ')⇒
            Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                    (set-error-ipc-maps caller partner σ σ' error-IPC msg)))
                    ,
                     snd(the(mbind S (abort_lift  ioprog)
                         (set-error-ipc-maps caller partner σ σ' error-IPC msg))))
          |  None ⇒ Some([], σ)))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using assms hyp0
  proof (cases a)
    fix aa b
    assume hyp1:a = (aa, b)
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases ioprog (IPC MAP (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by simp
    next
      case (Some ab)
      assume hyp2: ioprog (IPC MAP (SEND caller partner msg)) σ = Some ab
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac,ba)
        then show ?thesis
        using assms hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp4
         proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
```

case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))

```
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp11: mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-maps caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11
          proof (cases ad)
            fix ae bb
            assume hyp12: ad = (ae, bb)
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
            by simp
          qed
        qed
      qed
    qed
  qed
qed



lemma abort-map-send-obvious9:
  fst(the(mbind (IPC MAP (SEND caller partner msg)#S)(abort_lift  ioprog) σ))
=
      (if caller ∈ dom ( (th-flag σ))
         then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

     else (case ioprog (IPC MAP (SEND caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              NO-ERRORS#fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer
caller σ σ')))
           | Some(ERROR-MEM error-mem, σ')⇒
              ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)
                     (set-error-mem-maps caller partner σ σ' error-mem msg)))

           | Some(ERROR-IPC error-IPC, σ')⇒
              ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                     (set-error-ipc-maps caller partner σ σ' error-IPC msg)))

           | None ⇒ []))
  by (simp split: option.split errors.split,auto)
```

lemma *abort-map-recv-obvious0*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, (*error-tab-transfer caller* σ σ'))
using *assms*
by *simp*

lemma *abort-map-recv-obvious1*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-MEM error-mem*, (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
using *assms*
by *simp*

lemma *abort-map-recv-obvious2*:
assumes *not-in-err*: *caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-succes*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows $abort_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, (*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))
using *assms*
by *simp*

lemma *abort-map-recv-obvious3*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success* : *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*NO-ERRORS*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some a*
then show *?thesis*
using *assms hyp0*

proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious4*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))),
snd(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))))
using *assms*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*set-error-mem-mapr caller partner* σ σ' *error-mem msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:$a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious5*:
assumes *not-in-err* :*caller* $\notin$ *dom* ((*th-flag* σ))
and *ioprog-success*:*ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* ($abort_{lift}$ *ioprog*) (*set-error-ipc-mapr caller partner* σ σ' *error-IPC*

msg))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-doner caller partner* σ σ' *error-IPC msg*) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows *abort*$_{lift}$ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, σ)
using *assms*
by *simp*

lemma *abort-map-recv-obvious7*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
shows *mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*) (*abort*$_{lift}$ *ioprog*)
σ =
Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*

using *assms hyp0 hyp1*
by *simp*
qed
qed

lemma *abort-map-recv-obvious8*:
mbind ((*IPC MAP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ =
(*if caller* $\in$ *dom* ((*th-flag* σ))
then Some(*get-caller-error caller* σ#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))

else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ')
then Some(*NO-ERRORS*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))),
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ')
then Some(*ERROR-MEM error-mem*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ')
then Some(*ERROR-IPC error-IPC*#*fst*(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)))
,
snd(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))))
else if ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *None*
then Some([], σ)
else id (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*

using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa,b)$
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*) = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: $ab = (ac,ba)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp4*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ σ' *error-mem msg*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases ad*)
fix *ae bb*
assume *hyp5*: $ad = (ae, bb)$
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ'))
case *None*
then show *?thesis*
by *simp*
next
case (*Some af*)
assume *hyp6*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ σ') = *Some af*

```
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            proof (cases af)
              fix ag bc
              assume hyp7: af = (ag, bc)
              then show ?thesis
              using assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
              by simp
            qed
          qed
        qed
      qed
    qed
  qed
qed
```

```
lemma abort-map-recv-obvious8':
    mbind ((IPC MAP (RECV caller partner msg))#S)(abort_lift ioprog) σ =
      (if caller ∈ dom ( (th-flag σ))
       then Some(get-caller-error caller σ#fst(the(mbind S (abort_lift ioprog) σ)),
            snd(the(mbind S (abort_lift ioprog) σ)))
     else (case ioprog (IPC MAP (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
                        Some(NO-ERRORS#fst(the(mbind S (abort_lift ioprog)
(error-tab-transfer caller σ σ'))),
                             snd(the(mbind S (abort_lift ioprog) (error-tab-transfer
caller σ σ'))))
          | Some(ERROR-MEM error-mem, σ')⇒
            Some(ERROR-MEM error-mem#fst(the(mbind S (abort_lift ioprog)

                                 (set-error-mem-mapr caller partner σ σ' error-mem
msg)))
                    ,
                     snd(the(mbind S (abort_lift ioprog)
                                 (set-error-mem-mapr caller partner σ σ' error-mem
msg))))
          | Some(ERROR-IPC error-IPC, σ')⇒
             Some(ERROR-IPC error-IPC#fst(the(mbind S (abort_lift ioprog)
                                  (set-error-ipc-mapr caller partner σ σ' error-IPC
msg)))
                    ,
                     snd(the(mbind S (abort_lift ioprog)
                                  (set-error-ipc-mapr caller partner σ σ' error-IPC
msg))))
          |  None ⇒ Some([], σ)))
```

proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*:*a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by *simp*
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*,*ba*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp7*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7*
proof (*cases ad*)
fix *ae bb*
assume *hyp8*: *ad* = (*ae*, *bb*)
then show *?thesis*

using *assms hyp0 hyp1 hyp2 hyp3 hyp4 hyp7 hyp8*
by *simp*
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp5*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp9*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-mapr caller partner* σ *ba error-memory msg*) =
Some ad
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9*
proof (*cases ad*)
fix *ae bb*
assume *hyp10*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp5 hyp9 hyp10*
by *simp*
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp6*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp11*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-mapr caller partner* σ *ba error-IPC msg*) = *Some*
ad
then show *?thesis*
using *assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11*
proof (*cases ad*)
fix *ae bb*
assume *hyp12*: *ad* = (*ae*, *bb*)

```
            then show ?thesis
            using assms hyp0 hyp1 hyp2 hyp3 hyp6 hyp11 hyp12
            by simp
          qed
        qed
      qed
    qed
  qed
 qed
qed
```

```
lemma abort-map-recv-obvious9:
   fst(the(mbind ((IPC MAP (RECV caller partner msg))#S)(abort_lift  ioprog)
σ)) =
       (if caller ∈ dom ( (th-flag σ))
          then get-caller-error caller σ#fst(the(mbind S (abort_lift  ioprog) σ))

     else (case ioprog (IPC MAP (RECV caller partner msg)) σ of Some(NO-ERRORS,
σ')⇒
              NO-ERRORS#fst(the(mbind S (abort_lift  ioprog) (error-tab-transfer
caller σ σ')))
            | Some(ERROR-MEM error-mem, σ')⇒
                ERROR-MEM error-mem#fst(the(mbind S (abort_lift  ioprog)
                              (set-error-mem-mapr caller partner σ σ' error-mem
msg)))
            | Some(ERROR-IPC error-IPC, σ')⇒
                ERROR-IPC error-IPC#fst(the(mbind S (abort_lift  ioprog)
                                (set-error-ipc-mapr caller partner σ σ' error-IPC
msg)))
            |  None ⇒ []))
  by(simp split: option.split errors.split,auto)
```

L.6 Symbolic Execution Rules rules on DONE stage

```
lemma abort-done-send-obvious0:
  assumes not-in-err:
        caller ∉ dom ((th-flag) σ)
  assumes ioprog-success:ioprog (IPC DONE (SEND caller partner msg)) σ ≠
None
 shows  abort_lift  ioprog (IPC DONE (SEND caller partner msg)) σ = Some(NO-ERRORS,
σ)
  using assms
  by (simp split:option.split)

lemma abort-done-send-obvious1:
  assumes not-in-err:caller ∉ dom ((th-flag) σ)
```

and *exec-success*: *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *ioprog-success*:*ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) σ = *Some*(*out'*,σ')
shows $\sigma' = \sigma''$
using *assms*
by *auto*

lemma *abort-done-send-obvious2*:
assumes *not-in-err*:*caller* $\notin$ *dom* ((*th-flag*) σ)
and *exec-success*: *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *ioprog-success*:*ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq$ *None*
shows *mbind S* ($abort_{lift}$ *ioprog*) σ = *Some*(*out'*,σ') $\Longrightarrow$ *out''* = (*NO-ERRORS*#*out'*)
using *assms*
by *auto*

lemma *abort-done-send-obvious3*:
assumes *in-err*:*caller* $\in$ *dom* ((*th-flag*) σ)
shows $abort_{lift}$ *ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ =
Some(*get-caller-error caller* σ, *remove-caller-error caller* σ)
using *assms*
by *simp*

lemma *abort-done-send-obvious4*:
assumes *in-err*:*caller* $\in$ *dom* ((*th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
shows *hd out''* = *get-caller-error caller* σ
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp*, *elim conjE*, *simp add*: *HOL.eq-sym-conv*)
qed

qed

lemma *abort-done-send-obvious5*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (σ(|*th-flag* := (*th-flag* σ) (*caller* := *None*)|)) = *Some*(*out'*,σ')
shows *out''* = *the* (((*th-flag*) σ) *caller*) #*out'*
using *assms*
by *simp*

lemma *abort-done-send-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag* σ))
and *exec-success*: *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*: *mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some(*out'*,σ')
shows σ'' = σ'
using *assms*
by *simp*

lemma *abort-done-send-obvious7*:
assumes *exec-success* : *mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
Some (*out'*,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ ≠ *None*
shows(*if caller* ∈ *dom* ((*th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') ⇒ σ' = σ'')
else (*case mbind S* ($abort_{lift}$ *ioprog*) σ
of *Some* (*out''*,σ'') ⇒ σ' = σ''))
proof (*cases caller* ∈ *dom* ((*th-flag* σ)))
case *True*
assume *hyp0*: *caller* ∈ *dom* ((*th-flag* σ))
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some a

```
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by simp
    qed
  qed
next
  case False
  assume hyp0: caller ∉ dom ( (th-flag σ))
  then show ?thesis
  using assms hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    then show ?thesis
    using assms hyp0
    by simp
  next
    case (Some a)
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some a
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by auto
    qed
  qed
qed

lemma abort-done-send-obvious8:
  assumes execu-success : mbind ((IPC DONE (SEND caller partner msg))#S)(abort_lift
ioprog) σ =
                          Some (out',σ')
  and     ioprog-success: ioprog (IPC DONE (SEND caller partner msg)) σ ≠ None
  shows
      (if caller ∈ dom ((th-flag) σ)
        then (case mbind S (abort_lift ioprog)(remove-caller-error caller σ)
                  of Some (out'',σ'') ⇒ out' = (get-caller-error caller σ #out''))

        else (case mbind S (abort_lift ioprog) σ
              of   Some (out'',σ'') ⇒ out' = (NO-ERRORS#out'')))
proof (cases caller ∈ dom ( (th-flag σ)))
  case True
```

```
  assume hyp0: caller ∈ dom ( (th-flag σ))
  then show ?thesis
  using assms hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
    case None
    then show ?thesis
    by simp
  next
    case (Some a)
     assume hyp1: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller
σ) =
                  Some a
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by simp
    qed
  qed
next
  case False
  assume hyp0 : caller ∉ dom ( (th-flag σ))
  then show ?thesis
  using assms hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    then show ?thesis
    by simp
  next
    case (Some a)
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some a
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by auto
    qed
  qed
qed


lemma abort-done-send-obvious9:
      mbind ((IPC DONE (SEND caller partner msg))#S)(abort_lift  ioprog) σ =
```

```
      (if caller ∈ dom ((th-flag) σ)
      then Some (get-caller-error caller σ#
                 fst(the(mbind S (abort_lift ioprog)(remove-caller-error caller σ))),
                snd(the(mbind S (abort_lift ioprog) (remove-caller-error caller σ))))

       else (case ioprog (IPC DONE (SEND caller partner msg)) σ of None ⇒
Some ([], σ)
           | Some (out', σ') ⇒
             Some (NO-ERRORS# (fst o the)(mbind S (abort_lift ioprog) σ),
                  (snd o the)(mbind S (abort_lift ioprog) σ))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
             Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        by (simp add: split: option.split)
      qed
    qed
  qed
qed

lemma abort-done-send-obvious10:
      (fst o the)(mbind ((IPC DONE (SEND caller partner msg))#S)(abort_lift
ioprog) σ) =
```

```
        (if caller ∈ dom ((  th-flag) σ)
        then get-caller-error caller σ#
             (fst o the)(mbind S (abort_lift ioprog) (remove-caller-error caller σ))
        else
         (case ioprog (IPC DONE (SEND caller partner msg)) σ of
            None ⇒ []
          | Some (out', σ')⇒ NO-ERRORS# (fst o the)(mbind S (abort_lift ioprog)
σ)))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
                Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        by (simp split: option.split)
      qed
    qed
  qed
qed



lemma abort-done-recv-obvious0:
  assumes no-inerr:caller ∉ dom ((th-flag) σ)
  and     ioprog-success:ioprog (IPC DONE (RECV caller partner msg)) σ ≠ None
```

shows $abort_{lift}$ *ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ = *Some*(*NO-ERRORS*, σ)
using *assms*
by (*simp split*:*option.split*)

lemma *abort-done-recv-obvious1*:
assumes *not-in-err*:*caller* $\notin$ *dom* ((*th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*
shows *mbind S* ($abort_{lift}$ *ioprog*) σ = *Some*(*out'*,σ') $\Longrightarrow$ $\sigma' = \sigma''$
using *assms*
by *auto*

lemma *abort-done-recv-obvious2*:
assumes *not-inerr* :*caller* $\notin$ *dom* ((*th-flag*) σ)
and *exec-success* :*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*
shows *mbind S* ($abort_{lift}$ *ioprog*) σ = *Some*(*out'*,σ') $\Longrightarrow$ *out''* = (*NO-ERRORS*#*out'*)
using *assms*
by *auto*

lemma *abort-done-recv-obvious3*:
assumes *in-err*: *caller* $\in$ *dom* ((*th-flag*) σ)
shows $abort_{lift}$ *ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ =
Some(*get-caller-error caller* σ, *remove-caller-error caller* σ)
using *assms*
by *simp*

lemma *abort-done-recv-obvious4*:
assumes *in-err*:*caller* $\in$ *dom* ((*th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
shows *hd out''* = *get-caller-error caller* σ
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ) = *Some a*
then show *?thesis*
using *assms hyp0*
proof (*cases a*)

fix *aa b*
assume *hyp1*: $a = (aa, b)$
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp*, *elim conjE*, *simp add*: *HOL.eq-sym-conv*)
qed
qed

lemma *abort-done-recv-obvious5*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag*) σ)
and *exec-success*: *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) = *Some*(*out'*,σ')
shows *out''* = (*get-caller-error caller* σ #*out'*)
using *assms*
by *simp*

lemma *abort-done-recv-obvious6*:
assumes *in-err*:*caller* ∈ *dom* ((*th-flag*) σ)
and *exec-success*:*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*) ($abort_{lift}$ *ioprog*) σ =
Some(*out''*,σ'')
and *exec-success'*:*mbind S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some(*out'*,σ')
shows σ'' = σ'
using *assms*
by *simp*

lemma *abort-done-recv-obvious7*:
assumes *exec-success*: *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
Some (*out'*,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None*
shows (*if caller* ∈ *dom* ((*th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') ⇒ σ' = σ'')
else (*case mbind S* ($abort_{lift}$ *ioprog*) σ
of *Some* (*out''*,σ'') ⇒ σ' = σ''))
proof (*cases caller* ∈ *dom* ((*th-flag*) σ))
case *True*
assume *hyp0*: *caller* ∈ *dom* ((*th-flag*) σ)
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ))
case *None*
then show *?thesis*
using *assms hyp0*

by *simp*
next
case (*Some a*)
assume *hyp1*:$mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*remove-caller-error caller* σ) =
Some a
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *simp*
qed
qed
next
case *False*
assume *hyp0*: *caller* $\notin$ *dom* ((*th-flag*) σ)
then show *?thesis*
using *assms hyp0*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
using *assms hyp0*
by *simp*
next
case (*Some a*)
assume *hyp1*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *assms hyp0 hyp1*
proof (*cases a*)
fix *aa b*
assume *hyp2*: *a* = (*aa*, *b*)
then show *?thesis*
using *assms hyp0 hyp1 hyp2*
by *auto*
qed
qed
qed

lemma *abort-done-recv-obvious8*:
assumes *exec-success* : *mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*) σ =
Some (*out'*,σ')
and *ioprog-success*:*ioprog* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*
shows (*if caller* $\in$ *dom* ((*th-flag*) σ)
then (*case mbind S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller* σ)
of Some (*out''*,σ'') $\Rightarrow$ *out'* = (*get-caller-error caller* σ #*out''*))

```
        else (case mbind S (abort_lift ioprog) σ
              of   Some (out'',σ'') ⇒ out' = (NO-ERRORS#out'')))
proof (cases caller ∈ dom ( (th-flag σ)))
  case True
  assume hyp0: caller ∈ dom ( (th-flag σ))
  then show ?thesis
  using assms hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) (remove-caller-error caller σ))
    case None
    then show ?thesis
    using assms hyp0
    by simp
  next
    case (Some a)
     assume hyp1:mbind_FailSave S (abort_lift ioprog) (remove-caller-error caller
σ) =
                 Some a
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by simp
    qed
  qed
next
  case False
  assume hyp0: caller ∉ dom ( (th-flag σ))
  then show ?thesis
  using assms hyp0
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
    case None
    then show ?thesis
    using assms hyp0
    by simp
  next
    case (Some a)
    assume hyp1: mbind_FailSave S (abort_lift ioprog) σ = Some a
    then show ?thesis
    using assms hyp0 hyp1
    proof (cases a)
      fix aa b
      assume hyp2: a = (aa, b)
      then show ?thesis
      using assms hyp0 hyp1 hyp2
      by auto
    qed
```

```
  qed
qed

lemma abort-done-recv-obvious9:
      mbind ((IPC DONE (RECV caller partner msg))#S)(abort_lift  ioprog) σ =
        (if caller ∈ dom ((th-flag) σ)
        then Some ((get-caller-error caller σ#
                  fst(the(mbind S (abort_lift ioprog) (remove-caller-error caller σ)))),

                  snd(the(mbind S (abort_lift ioprog) (remove-caller-error caller σ))))

         else(case ioprog (IPC DONE (RECV caller partner msg)) σ of None ⇒
Some ([], σ)
             | Some (out', σ') ⇒
               Some (NO-ERRORS# (fst o the)(mbind S (abort_lift ioprog) σ),
                    (snd o the)(mbind S (abort_lift ioprog) σ))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
                Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        by (simp split: option.split)
      qed
    qed
```

qed
qed

lemma *abort-done-recv-obvious10*:
fst(*the*(*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*) σ)) =
(*if caller* $\in$ *dom* ((*th-flag*) σ)
then (*get-caller-error caller* σ#
fst(*the*(*mbind S* (*abort*$_{lift}$ *ioprog*) (*remove-caller-error caller* σ))))

else
(*case ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ *of*
None $\Rightarrow$ []
| *Some* (*out'*, σ')$\Rightarrow$ *NO-ERRORS*# (*fst o the*)(*mbind S* (*abort*$_{lift}$ *ioprog*) σ)))
by (*simp split*: *option.split*)

lemmas *trace-normalizer-errors-case* =
abort-prep-send-obvious9 abort-prep-recv-obvious9 abort-wait-send-obvious9
abort-wait-recv-obvious9 abort-buf-send-obvious9 abort-buf-recv-obvious9
abort-done-send-obvious10 abort-done-recv-obvious10

end

theory *IPC-symbolic-exec-rewriting*
imports *IPC-trace-normalizer*
begin

M Rewriting Rules for Symbolic Execution of Sequence Test Scheme

M.1 Symbolic Execution Rules for PREP stage

lemma *abort-prep-send-obvious10*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* $\in$ *dom* ((*th-flag*)σ)
then (σ $\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ') $\Rightarrow$
(*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*))
| *Some*(*ERROR-MEM error-mem*, σ')$\Rightarrow$
((*set-error-mem-preps caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem*

```
# outs)))
        | Some(ERROR-IPC error-IPC, σ')⇒
            ((set-error-ipc-preps caller partner σ σ' error-IPC msg)
             ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-IPC error-IPC#
outs)))

        |  None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC PREP (SEND caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
              assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
```

caller σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)

```
                    (set-error-ipc-preps caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
 qed
qed

lemma abort-prep-send-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC PREP (SEND caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
  (if caller ∈ dom ((th-flag)σ)
  then  (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs)))
  else (case ioprog (IPC PREP (SEND caller partner msg)) σ of
         Some(NO-ERRORS, σ') ⇒
        ((error-tab-transfer caller σ σ') ⊨
         (outs ← (mbind S(abort_lift  ioprog));P (NO-ERRORS # outs)))∧
        ((( th-flag) σ) caller = None) ∧
         (( th-flag) σ) caller =
         (( th-flag) (error-tab-transfer caller σ σ')) caller ∧
         (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
       | Some(ERROR-MEM error-mem, σ')⇒
       ((set-error-mem-preps caller partner σ σ' error-mem msg)
         ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem #
outs)))∧
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
         Some (ERROR-MEM error-mem))∧
     ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner
=
         Some (ERROR-MEM error-mem)) ∧
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
     (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
       | Some(ERROR-IPC error-IPC, σ')⇒
      ((set-error-ipc-preps caller partner σ σ' error-IPC msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-IPC error-IPC#
outs)))∧
```

```
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
         Some (ERROR-IPC error-IPC))∧
       ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
         Some (ERROR-IPC error-IPC)) ∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
       (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)

       |  None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC PREP (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC PREP (SEND caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
```

case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next

case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-preps caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious10''*:
($\sigma \models$ (*outs* ← (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
((*caller* ∈ *dom* ((*th-flag*)σ) ⟶
($\sigma \models$ (*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))))
∧
(*caller* ∉ *dom* ((*th-flag*)σ) ⟶
(*ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *None* ⟶ ($\sigma \models$ (*P* []))) ∧
((∀ *a* σ'.
(*a* = *NO-ERRORS* ⟶ *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some* (*NO-ERRORS*, σ') ⟶
((*error-tab-transfer caller* σ σ') ⊨
(*outs* ← (*mbind S*($abort_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-memory*. *a* = *ERROR-MEM error-memory* ⟶
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-MEM error-memory*, σ') ⟶
((*set-error-mem-preps caller partner* σ σ' *error-memory msg*)⊨
(*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) ∧
(∀ *error-IPC*. *a* = *ERROR-IPC error-IPC* ⟶
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, σ') ⟶
((*set-error-ipc-preps caller partner* σ σ' *error-IPC msg*)⊨
(*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))

proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*

next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed

```
      next
        case (ERROR-MEM error-memory)
        assume hyp4:ac = ERROR-MEM error-memory
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-preps caller partner σ ba error-memory msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-mem-preps caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-preps caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-preps caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
```

qed
qed
qed
qed
qed
qed

lemma *abort-prep-send-obvious10′*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error$
$caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\ a\ b.\ (a = NO\text{-}ERRORS \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (NO\text{-}ERRORS,\ b) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#$
$outs)))) \wedge$
$(\forall\ error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ b) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ error\text{-}memory,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}MEM$
$error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ b) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (\ ERROR\text{-}IPC$
$error\text{-}IPC\#\ outs)))))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ exec\text{-}action_{id}\text{-}Mon)\ \sigma)$
case *None*
then show *?thesis*
by *simp*

next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by(*simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *exec-action*$_{id}$*-Mon* (*IPC PREP* (*SEND caller partner msg*)) σ
= *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) *ba*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) *ba* =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-preps caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
PREP-SEND$_{id}$*-def*
split : *errors.split option.split split-if-asm*)

```
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                    (set-error-ipc-preps caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                         (set-error-ipc-preps caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              PREP-SEND_id-def
               split        : errors.split option.split split-if-asm)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-prep-send-obvious11:
 (σ ⊨ (outs ← (mbind ((IPC PREP (SEND caller partner msg))#S)
                     (abort_lift  exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (get-caller-error
caller σ # outs)))) ∧
   (caller ∉ dom (( th-flag)σ) ⟶
   (∀ a b. (exec-action_id-Mon-prep-fact0 caller partner σ msg ∧
          exec-action_id-Mon-prep-fact1 caller partner σ ⟶
     (σ(|current-thread := caller,
        thread-list    := update-th-ready caller (thread-list σ),
        error-codes    := NO-ERRORS,
        th-flag := th-flag σ|) ⊨
          (outs ← (mbind S(abort_lift  exec-action_id-Mon));P (NO-ERRORS #
```

$outs)))) \wedge$
$(\forall\, error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$((b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND|\!)\wedge$

$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg)$
$\wedge$
$error\text{-}memory = not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND)) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ error\text{-}memory,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}MEM$
$error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$((b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND|\!)\wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg\ IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND) \vee$
$(b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND|\!)\wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg\ IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND)) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (\ ERROR\text{-}IPC$
$error\text{-}IPC\#\ outs)))))))$

by (*auto simp add*: *abort-prep-send-obvious10'* $exec\text{-}action_{id}$*-Mon-prep-send-obvious3*
$exec\text{-}action_{id}$*-Mon-prep-send-obvious4* $exec\text{-}action_{id}$*-Mon-prep-send-obvious5*)

lemma *abort-prep-recv-obvious10*:

```
  (σ ⊨ (outs ← (mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom (( th-flag)σ)
    then (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs)))
    else (case ioprog (IPC PREP (RECV caller partner msg)) σ of
          Some(NO-ERRORS, σ') ⇒ (error-tab-transfer caller σ σ') ⊨
                              (outs ← (mbind (S)(abort_lift  ioprog)); P (NO-ERRORS
# outs))
        | Some(ERROR-MEM error-mem, σ')⇒
            ((set-error-mem-prepr caller partner σ σ' error-mem msg)
             ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem
# outs)))
        | Some(ERROR-IPC error-IPC, σ')⇒
            ((set-error-ipc-prepr caller partner σ σ' error-IPC msg)
             ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))
        |  None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC PREP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
```

assume *hyp4*: $ac = NO\text{-}ERRORS$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)$ (*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)

```
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                     (set-error-ipc-prepr caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-prepr caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-prep-recv-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom (( th-flag)σ)
    then (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs)))
    else (case ioprog (IPC PREP (RECV caller partner msg)) σ of
          Some(NO-ERRORS, σ') ⇒
         ((error-tab-transfer caller σ σ') ⊨
          (outs ← (mbind S(abort_lift  ioprog)); P (NO-ERRORS # outs)))∧
         ((( th-flag) σ) caller = None) ∧
          (( th-flag) σ) caller =
          (( th-flag) (error-tab-transfer caller σ σ')) caller ∧
           (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
        | Some(ERROR-MEM error-mem, σ')⇒
      ((set-error-mem-prepr caller partner σ σ' error-mem msg)
          ⊨ (outs ← (mbind S (abort_lift  ioprog)); P (ERROR-MEM error-mem
# outs)))∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
```

```
          Some (ERROR-MEM error-mem))∧
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner
=
          Some (ERROR-MEM error-mem)) ∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
      (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
          | Some(ERROR-IPC error-IPC, σ')⇒
          ((set-error-ipc-prepr caller partner σ σ' error-IPC msg)⊨
             (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))∧
          ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller
=
             Some (ERROR-IPC error-IPC))∧
        ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
             Some (ERROR-IPC error-IPC)) ∧
         ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller
=
        (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)
        |  None ⇒ (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC PREP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
```

using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed

```
      qed
    next
      case (ERROR-IPC error-IPC)
      assume hyp4:ac = ERROR-IPC error-IPC
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4
      proof (cases mbind_FailSave S (abort_lift ioprog)
                   (set-error-ipc-prepr caller partner σ ba error-IPC msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-prepr caller partner σ ba error-IPC msg) =
Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
          fix ae bb
          assume hyp6: ad = (ae, bb)
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
          by(simp add: valid-SE-def bind-SE-def)
        qed
      qed
    qed
  qed
 qed
qed
```

lemma *abort-prep-recv-obvious10''*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$((caller \in dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ [])))$
$\wedge$
$((\forall a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS, \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall error\text{-}memory.$

```
        a = ERROR-MEM error-memory ⟶
        ioprog (IPC PREP (RECV caller partner msg)) σ = Some (ERROR-MEM
error-memory, σ') ⟶
        ((set-error-mem-prepr caller partner σ σ' error-memory msg)⊨
          (outs ← (mbind (S)(abort_lift  ioprog)); P (ERROR-MEM error-memory
# outs)))) ∧
       (∀ error-IPC.
        a = ERROR-IPC error-IPC ⟶
         ioprog (IPC PREP (RECV caller partner msg)) σ = Some (ERROR-IPC
error-IPC, σ') ⟶
        ((set-error-ipc-prepr caller partner σ σ' error-IPC msg) ⊨
           (outs ← (mbind (S)(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs))))))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC PREP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
         proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
```

case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-prepr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-prepr caller partner* σ *ba error-IPC msg*))

```
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-prepr caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-prep-recv-obvious10':
 (σ ⊨ (outs ← (mbind ((IPC PREP (RECV caller partner msg))#S)(abort_lift
exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
  (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (get-caller-error caller
σ # outs)))) ∧
   (caller ∉ dom (( th-flag)σ) ⟶
   ((∀ a b.
    (a = NO-ERRORS ⟶
    exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ = Some (NO-ERRORS,
b) ⟶
     (σ(|current-thread := caller,
       thread-list    := update-th-ready caller (thread-list σ),
       error-codes    := NO-ERRORS,
       th-flag        := th-flag σ|) ⊨
          (outs ← (mbind S(abort_lift  exec-action_id-Mon));P (NO-ERRORS #
outs)))) ∧
    (∀ error-memory.
      a = ERROR-MEM error-memory ⟶
        exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ = Some
(ERROR-MEM error-memory, b) ⟶
       (σ(|current-thread := caller,
         thread-list    := update-th-current caller (thread-list σ),
         error-codes    := ERROR-MEM error-memory,
         state_id.th-flag := th-flag σ
```

```
                (caller ↦ (ERROR-MEM error-memory),
                 partner ↦ (ERROR-MEM error-memory))⦈
            ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (ERROR-MEM
error-memory # outs)))) ∧
    (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
          exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ = Some
(ERROR-IPC error-IPC, b) ⟶
        (σ⦇current-thread := caller,
          thread-list    := update-th-current caller (thread-list σ),
          error-codes    := ERROR-IPC error-IPC,
          state_id.th-flag := th-flag σ
                (caller ↦ (ERROR-IPC error-IPC),
                 partner ↦ (ERROR-IPC error-IPC))⦈
            ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P ( ERROR-IPC
error-IPC# outs))))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC PREP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
       assume hyp2: exec-action_id-Mon (IPC PREP (RECV caller partner msg))
σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
```

using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-RECV*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-RECV*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)

```
                (set-error-mem-prepr caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                 (set-error-mem-prepr caller partner σ ba error-memory msg)
= Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                          PREP-RECV_id-def
           split        : errors.split option.split split-if-asm)
      qed
    qed
  next
    case (ERROR-IPC error-IPC)
    assume hyp4:ac = ERROR-IPC error-IPC
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                (set-error-ipc-prepr caller partner σ ba error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                 (set-error-ipc-prepr caller partner σ ba error-IPC msg) =
Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                          PREP-RECV_id-def
           split        : errors.split option.split split-if-asm)
      qed
    qed
  qed
```

qed
qed
qed
qed

lemma *abort-prep-recv-obvious11*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$
$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$((\forall\ a\ b.$
$(a = NO\text{-}ERRORS \longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}memory.$
$a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$((b= \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV|\!) \wedge$

$\neg(list\text{-}all\ ((is\text{-}part\text{-}mem\text{-}th\ o\ the)\ ((thread\text{-}list\ \sigma)\ caller)\ (resource\ \sigma))msg) \wedge$
$error\text{-}memory = not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV)) \longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ error\text{-}memory,$
$state_{id}.th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$((b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV|\!) \wedge$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \wedge$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \wedge$
$\neg IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \wedge$
$error\text{-}IPC = error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV) \vee$
$(b = \sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$

$$
\begin{array}{l}
\quad\quad\quad\quad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC-23-in-PREP-RECV}\rrbracket \wedge \\
\quad\quad\quad \textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner } \sigma \textit{ msg} \wedge \\
\quad\quad\quad \neg \textit{IPC-params-c1 } ((\textit{the o thread-list } \sigma)\ \ \textit{partner}) \wedge \\
\quad\quad\quad \neg \textit{IPC-params-c2 } ((\textit{the o thread-list } \sigma)\ \ \textit{partner}) \wedge \\
\quad\quad\quad \textit{error-IPC} = \textit{error-IPC-23-in-PREP-RECV})) \longrightarrow \\
\quad\quad (\sigma\llparenthesis \textit{current-thread} := \textit{caller}, \\
\quad\quad\quad \textit{thread-list} \quad := \textit{update-th-current caller } (\textit{thread-list } \sigma), \\
\quad\quad\quad \textit{error-codes} \quad := \textit{ERROR-IPC error-IPC}, \\
\quad\quad\quad \textit{state}_{id}\textit{.th-flag} := \textit{th-flag } \sigma \\
\quad\quad\quad\quad\quad (\textit{caller} \mapsto (\textit{ERROR-IPC error-IPC}), \\
\quad\quad\quad\quad\quad\ \textit{partner} \mapsto (\textit{ERROR-IPC error-IPC}))\rrparenthesis \\
\quad\quad\quad\ \models (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \ \textit{exec-action}_{id}\textit{-Mon})); P\ (\ \textit{ERROR-IPC error-IPC\# outs})))))))
\end{array}
$$

by (*auto simp add*: *abort-prep-recv-obvious10′ exec-action*$_{id}$*-Mon-prep-recv-obvious3 exec-action*$_{id}$*-Mon-prep-recv-obvious4 exec-action*$_{id}$*-Mon-prep-recv-obvious5*)

M.2 Symbolic Execution Rules for WAIT stage

lemma *abort-wait-send-obvious10*:

$$
\begin{array}{l}
\quad (\sigma \models (\textit{outs} \leftarrow (\textit{mbind } ((\textit{IPC WAIT } (\textit{SEND caller partner msg}))\#S)(\textit{abort}_{lift}\ \textit{ioprog})); P\ \textit{outs})) = \\
\quad\quad (\textit{if caller} \in \textit{dom } ((\ \textit{th-flag})\sigma) \\
\quad\quad\ \ \textit{then } \ (\sigma \models (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \ \textit{ioprog})); P\ (\textit{get-caller-error caller } \sigma\ \# \ \textit{outs}))) \\
\quad\quad\ \ \textit{else } (\textit{case ioprog } (\textit{IPC WAIT } (\textit{SEND caller partner msg}))\ \sigma \textit{ of} \\
\quad\quad\quad\ \ \textit{Some}(\textit{NO-ERRORS}, \sigma') \Rightarrow (\textit{error-tab-transfer caller } \sigma\ \sigma') \\
\quad\quad\quad\quad\quad\quad\quad\quad \models (\textit{outs} \leftarrow (\textit{mbind } (S)(\textit{abort}_{lift}\ \ \textit{ioprog})); P\ (\textit{NO-ERRORS } \#\ \textit{outs})) \\
\quad\quad\quad\ \ | \ \textit{Some}(\textit{ERROR-MEM error-mem}, \sigma') \Rightarrow \\
\quad\quad\quad\quad\ \ ((\textit{set-error-mem-waits caller partner } \sigma\ \sigma'\ \textit{error-mem msg}) \\
\quad\quad\quad\quad\ \ \models (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \ \textit{ioprog})); P\ (\textit{ERROR-MEM error-mem}\ \# \ \textit{outs}))) \\
\quad\quad\quad\ \ | \ \textit{Some}(\textit{ERROR-IPC error-IPC}, \sigma') \Rightarrow \\
\quad\quad\quad\quad\ \ ((\textit{set-error-ipc-waits caller partner } \sigma\ \sigma'\ \textit{error-IPC msg}) \\
\quad\quad\quad\quad\ \ \models (\textit{outs} \leftarrow (\textit{mbind } S(\textit{abort}_{lift}\ \ \textit{ioprog})); P\ (\ \textit{ERROR-IPC error-IPC}\ \# \ \textit{outs}))) \\
\\
\quad\quad\quad\ \ | \ \textit{None} \Rightarrow (\sigma \models (P\ [])))))
\end{array}
$$

proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
  case *None*
  then show *?thesis*
  by *simp*
next
  case (*Some a*)
  assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
  then show *?thesis*
  using *hyp0*
  proof (*cases a*)
    fix *aa b*

assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))

```
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-mem-waits caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-waits caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-waits caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
 qed
qed
```

lemma *abort-wait-send-obvious12*:
(σ ⊨ (outs ← (mbind ((IPC WAIT (SEND caller partner msg))#S)(abort$_{lift}$ ioprog)); P outs)) =
(if caller ∈ dom ((th-flag)σ)
then (σ ⊨ (outs ← (mbind S(abort$_{lift}$ ioprog)); P (get-caller-error caller σ # outs)))
else (case ioprog (IPC WAIT (SEND caller partner msg)) σ of
Some(NO-ERRORS, σ′) ⇒
((error-tab-transfer caller σ σ′)⊨
(outs ← (mbind S (abort$_{lift}$ ioprog)); P (NO-ERRORS # outs)))∧
(((th-flag) σ) caller = None) ∧
((th-flag) σ) caller =
((th-flag) (error-tab-transfer caller σ σ′)) caller ∧
(th-flag σ = th-flag (error-tab-transfer caller σ σ′))
| Some(ERROR-MEM error-mem, σ′)⇒
((set-error-mem-waits caller partner σ σ′ error-mem msg) ⊨
(outs ← (mbind S(abort$_{lift}$ ioprog)); P (ERROR-MEM error-mem # outs)))∧
(((th-flag) (set-error-mem-maps caller partner σ σ′ error-mem msg)) caller =
Some (ERROR-MEM error-mem))∧
(((th-flag) (set-error-mem-maps caller partner σ σ′ error-mem msg)) partner =
Some (ERROR-MEM error-mem)) ∧
(((th-flag) (set-error-mem-maps caller partner σ σ′ error-mem msg)) caller =
((th-flag) (set-error-mem-maps caller partner σ σ′ error-mem msg)) partner)
| Some(ERROR-IPC error-IPC, σ′)⇒
((set-error-ipc-waits caller partner σ σ′ error-IPC msg) ⊨
(outs ← (mbind S(abort$_{lift}$ ioprog)); P (ERROR-IPC error-IPC# outs)))∧
(((th-flag) (set-error-ipc-maps caller partner σ σ′ error-IPC msg)) caller =
Some (ERROR-IPC error-IPC))∧
(((th-flag) (set-error-ipc-maps caller partner σ σ′ error-IPC msg)) partner =
Some (ERROR-IPC error-IPC)) ∧
(((th-flag) (set-error-ipc-maps caller partner σ σ′ error-IPC msg)) caller =
((th-flag) (set-error-ipc-maps caller partner σ σ′ error-IPC msg)) partner)
| None ⇒ (σ ⊨ (P []))))
proof (*cases mbind*$_{FailSave}$ *S* (abort$_{lift}$ ioprog) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (abort$_{lift}$ ioprog) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)

fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)

```
                    (set-error-mem-waits caller partner σ ba error-memory msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-mem-waits caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by (simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-waits caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-waits caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by (simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed
```

lemma *abort-wait-send-obvious10''*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
((*caller* ∈ *dom* ((*th-flag*)σ) ⟶
(σ ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*))))
∧
(*caller* ∉ *dom* ((*th-flag*)σ) ⟶
(*ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *None* ⟶ (σ ⊨ (*P* []))) ∧
((∀ *a* σ'.
(*a* = *NO-ERRORS* ⟶
ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*NO-ERRORS*, σ') ⟶
((*error-tab-transfer caller* σ σ') ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-memory*. *a* = *ERROR-MEM error-memory* ⟶
ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*ERROR-MEM error-memory*, σ') ⟶
((*set-error-mem-waits caller partner* σ σ' *error-memory msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) ∧
(∀ *error-IPC*. *a* = *ERROR-IPC error-IPC* ⟶
ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, σ') ⟶
((*set-error-ipc-waits caller partner* σ σ' *error-IPC msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)

assume *hyp2*: *ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*

proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed

lemma *abort-wait-send-obvious10'*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*)) =
((*caller* $\in$ *dom* ((*th-flag*)σ) $\longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*)))) $\wedge$
(*caller* $\notin$ *dom* ((*th-flag*)σ) $\longrightarrow$
($\forall$ *a b*.
(*a* = *NO-ERRORS* $\longrightarrow$
exec-action$_{id}$-*Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*

```
(NO-ERRORS, b) ⟶
      ((σ(|current-thread := caller,
          thread-list    := update-th-waiting caller (thread-list σ),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ|))
         ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))) ∧
    (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
          exec-action_id-Mon (IPC WAIT (SEND caller partner msg)) σ = Some
(ERROR-IPC error-IPC, b) ⟶
      ((σ(|current-thread := caller,
          thread-list    := update-th-current caller (thread-list σ),
          error-codes    := ERROR-IPC error-IPC,
          state_id.th-flag := th-flag σ
          (caller ↦ (ERROR-IPC error-IPC),
           partner ↦ (ERROR-IPC error-IPC))|))
             ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P ( ERROR-IPC
error-IPC# outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC WAIT (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
     assume hyp2: exec-action_id-Mon (IPC WAIT (SEND caller partner msg)) σ
= Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
```

case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *WAIT-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *PREP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next

case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waits caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*
WAIT-SEND$_{id}$*-def*
split : *errors.split option.split option.split-asm split-if-asm*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-ipc-waits caller partner* σ *ba error-IPC msg*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *exec-action*$_{id}$*-Mon-def valid-SE-def bind-SE-def*

```
                    WAIT-SEND_id-def
          split      : errors.split option.split option.split-asm split-if-asm)
      qed
    qed
  qed
 qed
qed

lemma abort-wait-send-obvious11:
 (σ ⊨ (outs ← (mbind ((IPC WAIT (SEND caller partner msg))#S)
                     (abort_lift  exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
    (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                 P (get-caller-error caller σ # outs)))) ∧
   (caller ∉ dom (( th-flag)σ) ⟶
    (∀ a b. (IPC-send-comm-check-st_id caller partner σ ∧
          IPC-params-c4 caller partner ∧ IPC-params-c5 partner σ  ⟶
      ((σ⦇current-thread := caller,
          thread-list    := update-th-waiting caller (thread-list σ),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ⦈)
        ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))) ∧
     (∀ error-IPC.
     (
     ¬ IPC-send-comm-check-st_id caller partner σ ⟶
    (σ⦇current-thread := caller ,
      thread-list    := update-th-current caller (thread-list σ),
      error-codes    := ERROR-IPC error-IPC-1-in-WAIT-SEND,
      th-flag        := th-flag σ
      (caller  ↦ (ERROR-IPC error-IPC-1-in-WAIT-SEND),
       partner ↦ (ERROR-IPC error-IPC-1-in-WAIT-SEND))⦈ ⊨
      (outs ← (mbind S(abort_lift  exec-action_id-Mon));
              P ( ERROR-IPC error-IPC-1-in-WAIT-SEND# outs)))) ∧
     (a = ERROR-IPC error-IPC ⟶
      IPC-send-comm-check-st_id caller partner σ ⟶
      ((¬ IPC-params-c4   caller  partner  ⟶
        b = σ⦇current-thread := caller ,
             thread-list := update-th-current caller (thread-list σ),
             error-codes := ERROR-IPC error-IPC-3-in-WAIT-SEND⦈ ∧
        error-IPC = error-IPC-3-in-WAIT-SEND) ∧
       (IPC-params-c4   caller  partner  ⟶
       ((¬ IPC-params-c5 partner σ ⟶
          b = update-state-wait-send-params5 σ caller ∧
           error-codes (update-state-wait-send-params5 σ caller) = ERROR-IPC
error-IPC ) ∧
       ¬ IPC-params-c5 partner σ ))) ⟶
```

```
      ((σ(|current-thread := caller,
          thread-list    := update-th-current caller (thread-list σ),
          error-codes    := ERROR-IPC error-IPC,
          th-flag        := th-flag σ
          (caller ↦ (ERROR-IPC error-IPC),
           partner ↦ (ERROR-IPC error-IPC))|))
              ⊨ (outs ← (mbind S(abort_lift exec-action_id-Mon));
                          P ( ERROR-IPC error-IPC# outs))))))))
 by (auto simp add: abort-wait-send-obvious10' exec-action_id-Mon-wait-send-obvious3
                    exec-action_id-Mon-wait-send-obvious4)



lemma abort-wait-recv-obvious10:
  (σ ⊨ (outs ← (mbind ((IPC WAIT (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom ((th-flag)σ)
    then (σ ⊨ (outs ← (mbind S(abort_lift ioprog)); P (get-caller-error caller σ
# outs)))
    else (case ioprog (IPC WAIT (RECV caller partner msg)) σ of
          Some(NO-ERRORS, σ') ⇒
          (error-tab-transfer caller σ σ') ⊨
          (outs ← (mbind S(abort_lift ioprog));P (NO-ERRORS # outs))
        | Some(ERROR-MEM error-mem, σ')⇒
          ((set-error-mem-waitr caller partner σ σ' error-mem msg)
            ⊨ (outs ← (mbind S(abort_lift ioprog)); P (ERROR-MEM error-mem
# outs)))
        | Some(ERROR-IPC error-IPC, σ')⇒
          ((set-error-ipc-waitr caller partner σ σ' error-IPC msg)
            ⊨ (outs ← (mbind S(abort_lift ioprog)); P ( ERROR-IPC error-IPC#
outs)))
        | None ⇒ (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC WAIT (RECV caller partner msg)) σ)
      case None
      then show ?thesis
```

using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)

(*set-error-mem-waitr caller partner σ ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner σ ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner σ ba error-IPC msg*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious12*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* ∈ *dom* ((*th-flag*)*σ*)
then (*σ* ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ # outs*)))

```
    else (case ioprog (IPC WAIT (RECV caller partner msg)) σ of
          Some(NO-ERRORS, σ') ⇒
        ((error-tab-transfer caller σ σ') ⊨
         (outs ← (mbind S(abort_lift  ioprog));P (NO-ERRORS # outs)))∧
        ((( th-flag) σ) caller = None) ∧
         (( th-flag) σ) caller =
         (( th-flag) (error-tab-transfer caller σ σ')) caller ∧
         (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
          | Some(ERROR-MEM error-mem, σ')⇒
        ((set-error-mem-waitr caller partner σ σ' error-mem msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem #
outs)))∧
        ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
          Some (ERROR-MEM error-mem))∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner
=
          Some (ERROR-MEM error-mem)) ∧
        ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
       (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
          | Some(ERROR-IPC error-IPC, σ')⇒
        ((set-error-ipc-waitr caller partner σ σ' error-IPC msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))∧
        ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
          Some (ERROR-IPC error-IPC))∧
         ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
          Some (ERROR-IPC error-IPC)) ∧
        ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
         (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)
          | None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC WAIT (RECV caller partner msg)) σ)
      case None
```

then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by (*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)

assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-waitr caller partner* σ *ba error-IPC msg*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-wait-recv-obvious10''*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
((*caller* $\in$ *dom* ((*th-flag*)σ) $\longrightarrow$

$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))))$
$\wedge$
$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = None \longrightarrow (\sigma \models (P\ [])))$
$\wedge$
$((\forall\ a\ \sigma'.$
$(a = NO\text{-}ERRORS \longrightarrow ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some\ (NO\text{-}ERRORS, \sigma') \longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}memory.\ a = ERROR\text{-}MEM\ error\text{-}memory \longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}MEM$
$error\text{-}memory, \sigma') \longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}memory\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM$
$error\text{-}memory\ \#\ outs)))) \wedge$
$(\forall\ error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC \longrightarrow$
$ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC$
$error\text{-}IPC, \sigma') \longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#$
$outs))))))))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof $(cases\ a)$
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof $(cases\ ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma)$
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by $(simp\ add$: *valid-SE-def bind-SE-def*$)$
next
case $(Some\ ab)$
assume *hyp2*: $ioprog\ (IPC\ WAIT\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ ab$
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof $(cases\ ab)$
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)

```
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                     (set-error-ipc-waitr caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-waitr caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-wait-recv-obvious10':
  (σ ⊨ (outs ← (mbind ((IPC WAIT (RECV caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                  P (get-caller-error caller σ # outs)))) ∧
    (caller ∉ dom (( th-flag)σ) ⟶
     (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC WAIT (RECV caller
partner msg)) σ =
                              Some (NO-ERRORS, b) ⟶
       ((σ(|current-thread := caller,
          thread-list    := update-th-waiting caller (thread-list σ),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ |))
         ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))) ∧
```

```
      (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
         exec-action_id-Mon (IPC WAIT (RECV caller partner msg)) σ = Some
(ERROR-IPC error-IPC, b) ⟶
       ((σ(|current-thread := caller,
          thread-list    := update-th-current caller (thread-list σ),
          error-codes    := ERROR-IPC error-IPC,
          state_id.th-flag := state_id.th-flag σ
          (caller ↦ (ERROR-IPC error-IPC),
           partner ↦ (ERROR-IPC error-IPC))|))
              ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                        P ( ERROR-IPC error-IPC# outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC WAIT (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
       assume hyp2: exec-action_id-Mon (IPC WAIT (RECV caller partner msg))
σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
       proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) (error-tab-transfer
caller σ ba))
           case None
```

then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *WAIT-RECV*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-waitr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next

```
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-mem-waitr caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              WAIT-RECV_id-def
              split       : errors.split option.split option.split-asm split-if-asm)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                     (set-error-ipc-waitr caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-ipc-waitr caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              WAIT-RECV_id-def
              split       : errors.split option.split option.split-asm split-if-asm)
          qed
        qed
      qed
    qed
  qed
 qed
qed
```

lemma *abort-wait-recv-obvious11*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*)) =
((*caller* ∈ *dom* ((*th-flag*)σ) ⟶
(σ ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*)))) ∧
(*caller* ∉ *dom* ((*th-flag*)σ) ⟶
(∀ *a b*. (*IPC-recv-comm-check-st*$_{id}$ *caller partner* σ ∧
IPC-params-c4 caller partner ∧ *IPC-params-c5 partner* σ ⟶
((σ⦇*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ⦈)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-IPC*.
(
¬ *IPC-recv-comm-check-st*$_{id}$ *caller partner* σ ⟶
(σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-RECV*,
th-flag := *th-flag* σ
(*caller* ↦ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*),
partner ↦ (*ERROR-IPC error-IPC-1-in-WAIT-RECV*))⦈ ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-RECV*# *outs*)))) ∧
(*a* = *ERROR-IPC error-IPC* ⟶
IPC-recv-comm-check-st$_{id}$ *caller partner* σ ⟶
((¬ *IPC-params-c4 caller partner* ⟶
b = σ⦇*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-3-in-WAIT-RECV*⦈ ∧
error-IPC = *error-IPC-3-in-WAIT-RECV*) ∧
(*IPC-params-c4 caller partner* ⟶
((¬ *IPC-params-c5 partner* σ ⟶
b = *update-state-wait-recv-params5* σ *caller* ∧
error-codes (*update-state-wait-recv-params5* σ *caller*) = *ERROR-IPC error-IPC*) ∧
¬ *IPC-params-c5 partner* σ))) ⟶
((σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(*caller* ↦ (*ERROR-IPC error-IPC*),
partner ↦ (*ERROR-IPC error-IPC*))⦈)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC*# *outs*))))))))
by (*auto simp add*: *abort-wait-recv-obvious10′ exec-action*$_{id}$-*Mon-wait-recv-obvious3*
exec-action$_{id}$-*Mon-wait-recv-obvious4*)

M.3 Symbolic Execution Rules for BUF stage

lemma *abort-buf-send-obvious10*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* ∈ *dom* ((*th-flag* σ))
then (σ ⊨ (*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
else (*case ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ *of*
Some(*NO-ERRORS*, σ′) ⇒
(*error-tab-transfer caller* σ σ′) ⊨
(*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))
| *Some*(*ERROR-MEM error-mem*, σ′)⇒
((*set-error-mem-bufs caller partner* σ σ′ *error-mem msg*)
⊨ (*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*)))
| *Some*(*ERROR-IPC error-IPC*, σ′)⇒
((*set-error-ipc-bufs caller partner* σ σ′ *error-IPC msg*)
⊨ (*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))
| *None* ⇒ (σ ⊨ (*P* [])))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) σ = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*

```
      proof (cases ac)
        case NO-ERRORS
        assume hyp4: ac = NO-ERRORS
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
       proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
            assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ ba) = Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-MEM error-memory)
        assume hyp4:ac = ERROR-MEM error-memory
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-bufs caller partner σ ba error-memory msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                     (set-error-mem-bufs caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
```

```
        next
          case (ERROR-IPC error-IPC)
          assume hyp4:ac = ERROR-IPC error-IPC
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-bufs caller partner σ ba error-IPC msg))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
            assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some
ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
            proof (cases ad)
              fix ae bb
              assume hyp6: ad = (ae, bb)
              then show ?thesis
              using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
              by(simp add: valid-SE-def bind-SE-def)
            qed
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-buf-send-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC BUF (SEND caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom ( (th-flag σ))
    then (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs)))
    else (case ioprog (IPC BUF (SEND caller partner msg)) σ of
           Some(NO-ERRORS, σ') ⇒
       ((error-tab-transfer caller σ σ') ⊨
        (outs ← (mbind S(abort_lift  ioprog)); P (NO-ERRORS # outs)))∧
       ((( th-flag) σ) caller = None) ∧
        (( th-flag) σ) caller =
        (( th-flag) (error-tab-transfer caller σ σ')) caller ∧
        (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
          | Some(ERROR-MEM error-mem, σ')⇒
       ((set-error-mem-bufs caller partner σ σ' error-mem msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem #
outs)))∧
```

```
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
         Some (ERROR-MEM error-mem))∧
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner
=
         Some (ERROR-MEM error-mem)) ∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
      (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
         | Some(ERROR-IPC error-IPC, σ')⇒
       ((set-error-ipc-bufs caller partner σ σ' error-IPC msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
         Some (ERROR-IPC error-IPC))∧
       ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
         Some (ERROR-IPC error-IPC)) ∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
       (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)
         | None ⇒ (σ ⊨ (P [])))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC BUF (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC BUF (SEND caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
```

using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed

```
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                     (set-error-ipc-bufs caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-buf-send-obvious10'':
  (σ ⊨ (outs ← (mbind ((IPC BUF (SEND caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
  (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ # outs))))
∧
   (caller ∉ dom (( th-flag)σ) ⟶
   (ioprog (IPC BUF (SEND caller partner msg)) σ = None ⟶ (σ ⊨ (P []))) ∧
   ((∀ a σ'.
     (a = NO-ERRORS ⟶ ioprog (IPC BUF (SEND caller partner msg)) σ =
Some (NO-ERRORS, σ') ⟶
    ((error-tab-transfer caller σ σ') ⊨
     (outs ← (mbind S(abort_lift  ioprog));P (NO-ERRORS # outs)))) ∧
    (∀ error-memory. a = ERROR-MEM error-memory ⟶
       ioprog (IPC BUF (SEND caller partner msg)) σ = Some (ERROR-MEM
error-memory, σ') ⟶
      ((set-error-mem-bufs caller partner σ σ' error-memory msg)
```

```
        ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-memory
# outs)))) ∧
     (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
        ioprog (IPC BUF (SEND caller partner msg)) σ = Some (ERROR-IPC
error-IPC, σ') ⟶
       ((set-error-ipc-bufs caller partner σ σ' error-IPC msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs))))))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC BUF (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC BUF (SEND caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
```

assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufs caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)

```
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-buf-send-obvious10':
  (σ ⊨ (outs ← (mbind ((IPC BUF (SEND caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                  P (get-caller-error caller σ # outs)))) ∧
    (caller ∉ dom (( th-flag)σ) ⟶
     (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC BUF (SEND caller
partner msg)) σ =
                              Some (NO-ERRORS, b) ⟶
       ((σ⦇current-thread := caller,
          resource       := foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
                              (zip (get-th-addrs partner σ) (get-msg-values msg σ)),
          thread-list  := update-th-ready caller
                          (update-th-ready partner
                          (thread-list σ)),
          error-codes  := NO-ERRORS,
          th-flag      := th-flag σ⦈)
         ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))) ∧
      (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
        exec-action_id-Mon (IPC BUF (SEND caller partner msg)) σ = Some
(ERROR-IPC error-IPC, b) ⟶
       ((σ⦇current-thread := caller,
          thread-list    := update-th-current caller (thread-list σ),
          error-codes    := ERROR-IPC error-IPC,
          state_id.th-flag := state_id.th-flag σ
          (caller  ↦ (ERROR-IPC error-IPC),
           partner ↦ (ERROR-IPC error-IPC))⦈)
             ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
```

```
                                P ( ERROR-IPC error-IPC# outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC BUF (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: exec-action_id-Mon (IPC BUF (SEND caller partner msg)) σ
= Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) ba)
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
              assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon) ba =
Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
            proof (cases ad)
              fix ae bb
              assume hyp6: ad = (ae, bb)
```

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: $BUF\text{-}SEND_{id}$*-def valid-SE-def bind-SE-def* $exec\text{-}action_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def* $exec\text{-}action_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def* $exec\text{-}action_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ $exec\text{-}action_{id}$*-Mon*)
(*set-error-mem-bufs caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*

```
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              BUF-SEND_id-def
               split        : errors.split option.split list.split-asm split-if-asm)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                     (set-error-ipc-bufs caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-ipc-bufs caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              BUF-SEND_id-def
               split        : errors.split option.split list.split-asm split-if-asm)
          qed
        qed
      qed
    qed
  qed
 qed
qed

lemma abort-buf-send-obvious11:
  (σ ⊨ (outs ← (mbind ((IPC BUF (SEND caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                  P (get-caller-error caller σ # outs)))) ∧
    (caller ∉ dom (( th-flag)σ) ⟶
     (∀ a b.
      (a = NO-ERRORS ⟶ IPC-buf-check-st_id caller partner σ ⟶
       ((σ(|current-thread := caller,
          resource       := foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
```

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad (zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),\\
\quad thread\text{-}list \quad := update\text{-}th\text{-}ready\ caller\\
\qquad\qquad\qquad\quad (update\text{-}th\text{-}ready\ partner\\
\qquad\qquad\qquad\quad (thread\text{-}list\ \sigma)),\\
\quad error\text{-}codes \quad := NO\text{-}ERRORS,\\
\quad th\text{-}flag \qquad := th\text{-}flag\ \sigma\rbag)\\
\quad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\\
outs)))) \wedge
\end{array}
$$

$$
\begin{array}{l}
(a = NO\text{-}ERRORS \wedge msg = [] \longrightarrow IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow\\
\quad ((\sigma\lbag current\text{-}thread := caller,\\
\quad resource \qquad := \ resource\ \sigma,\\
\quad thread\text{-}list \quad := update\text{-}th\text{-}ready\ caller\\
\qquad\qquad\qquad\quad (update\text{-}th\text{-}ready\ partner\\
\qquad\qquad\qquad\quad (thread\text{-}list\ \sigma)),\\
\quad error\text{-}codes \quad := NO\text{-}ERRORS,\\
\quad th\text{-}flag \qquad := th\text{-}flag\ \sigma\rbag)\\
\quad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\\
outs)))) \wedge
\end{array}
$$

$$
\begin{array}{l}
(a = \ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND \longrightarrow\\
\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow\\
\quad ((\sigma\lbag current\text{-}thread := caller,\\
\quad thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),\\
\quad error\text{-}codes \quad := \ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND,\\
\quad state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma\\
\quad (caller\ \mapsto (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND),\\
\quad partner \mapsto (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND))\rbag)\\
\qquad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\\
\qquad\qquad P\ (\ \ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\#\ outs)))))))
\end{array}
$$

by (*simp add*: *abort-buf-send-obvious10′ exec-action*$_{id}$*-Mon-buf-send-obvious3*, *auto*)

lemma *abort-buf-recv-obvious10*:

$$
\begin{array}{l}
(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\\
ioprog));\ P\ outs)) =\\
\quad (if\ caller \in dom\ ((\ th\text{-}flag)\sigma)\\
\quad\ then\ \ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\\
\#\ outs)))\\
\quad\ else\ (case\ ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma\ of\\
\qquad Some(NO\text{-}ERRORS, \sigma') \Rightarrow\\
\qquad (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models\\
\qquad (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));P\ (NO\text{-}ERRORS\ \#\ outs))\\
\qquad |\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma')\Rightarrow\\
\qquad\quad ((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)\\
\qquad\quad \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\\
\#\ outs)))\\
\qquad |\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma')\Rightarrow
\end{array}
$$

```
          ((set-error-ipc-bufr caller partner σ σ' error-IPC msg) ⊨
           (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC# outs)))
       | None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC BUF (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC BUF (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
              assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller σ ba) = Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
```

proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*

```
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-buf-recv-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC BUF (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
  (if caller ∈ dom (( th-flag)σ)
   then  (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ
# outs)))
   else (case ioprog (IPC BUF (RECV caller partner msg)) σ of
          Some(NO-ERRORS, σ') ⇒
          ((error-tab-transfer caller σ σ') ⊨
           (outs ← (mbind S(abort_lift  ioprog));P (NO-ERRORS # outs)))∧
           ((( th-flag) σ) caller = None) ∧
            (( th-flag) σ) caller =
            (( th-flag) (error-tab-transfer caller σ σ')) caller ∧
            (th-flag σ = th-flag (error-tab-transfer caller σ σ'))
        | Some(ERROR-MEM error-mem, σ')⇒
      ((set-error-mem-bufr caller partner σ σ' error-mem msg)
        ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem #
outs)))∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
          Some (ERROR-MEM error-mem))∧
      ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner
=
          Some (ERROR-MEM error-mem)) ∧
       ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
      (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) partner)
        | Some(ERROR-IPC error-IPC, σ')⇒
       ((set-error-ipc-bufr caller partner σ σ' error-IPC msg) ⊨
      (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC# outs)))∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
          Some (ERROR-IPC error-IPC))∧
       ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
```

```
        Some (ERROR-IPC error-IPC)) ∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
       (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner)
        | None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC BUF (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC BUF (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift ioprog) (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
              assume hyp5: mbind_FailSave S (abort_lift ioprog) (error-tab-transfer
caller σ ba) = Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
```

proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-bufr caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-bufr caller partner* σ *ba error-IPC msg*) = *Some ad*
then show *?thesis*

```
          using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-buf-recv-obvious10'':
 (σ ⊨ (outs ← (mbind ((IPC BUF (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
  (σ ⊨ (outs ← (mbind S(abort_lift ioprog)); P (get-caller-error caller σ # outs))))
∧
   (caller ∉ dom (( th-flag)σ) ⟶
   (ioprog (IPC BUF (RECV caller partner msg)) σ = None ⟶ (σ ⊨ (P []))) ∧
  ((∀ a σ'.
     (a = NO-ERRORS ⟶ ioprog (IPC BUF (RECV caller partner msg)) σ =
                                Some (NO-ERRORS, σ') ⟶
     ((error-tab-transfer caller σ σ') ⊨
      (outs ← (mbind S(abort_lift ioprog));P (NO-ERRORS # outs)))) ∧
   (∀ error-memory. a = ERROR-MEM error-memory ⟶
      ioprog (IPC BUF (RECV caller partner msg)) σ = Some (ERROR-MEM
error-memory, σ') ⟶
     ((set-error-mem-bufr caller partner σ σ' error-memory msg)
      ⊨ (outs ← (mbind S(abort_lift ioprog)); P (ERROR-MEM error-memory #
outs)))) ∧
   (∀ error-IPC. a = ERROR-IPC error-IPC ⟶
       ioprog (IPC BUF (RECV caller partner msg)) σ = Some (ERROR-IPC
error-IPC, σ') ⟶
     ((set-error-ipc-bufr caller partner σ σ' error-IPC msg)
       ⊨ (outs ← (mbind S(abort_lift ioprog)); P ( ERROR-IPC error-IPC#
outs))))))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
```

proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae,\ bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*

```
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-mem-bufr caller partner σ ba error-memory msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-mem-bufr caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-bufr caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-bufr caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed
```

qed

lemma *abort-buf-recv-obvious10'*:
(σ ⊨ (outs ← (mbind ((IPC BUF (RECV caller partner msg))#S)
(abort$_{lift}$ exec-action$_{id}$-Mon)); P outs)) =
((caller ∈ dom ((th-flag)σ) ⟶
(σ ⊨ (outs ← (mbind S(abort$_{lift}$ exec-action$_{id}$-Mon));
P (get-caller-error caller σ # outs)))) ∧
(caller ∉ dom ((th-flag)σ) ⟶
(∀ a b. (a = NO-ERRORS ⟶ exec-action$_{id}$-Mon (IPC BUF (RECV caller partner msg)) σ =
Some (NO-ERRORS, b) ⟶
((σ(|current-thread := caller,
resource := foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
(zip (get-th-addrs caller σ) (get-msg-values msg σ)),
thread-list := update-th-ready caller
(update-th-ready partner
(thread-list σ)),
error-codes := NO-ERRORS,
th-flag := th-flag σ|))
⊨ (outs ← (mbind S(abort$_{lift}$ exec-action$_{id}$-Mon)); P (NO-ERRORS # outs)))) ∧
(∀ error-IPC. a = ERROR-IPC error-IPC ⟶
exec-action$_{id}$-Mon (IPC BUF (RECV caller partner msg)) σ = Some (ERROR-IPC error-IPC, b) ⟶
((σ(|current-thread := caller,
thread-list := update-th-current caller (thread-list σ),
error-codes := ERROR-IPC error-IPC,
state$_{id}$.th-flag := state$_{id}$.th-flag σ
(caller ↦ (ERROR-IPC error-IPC),
partner ↦ (ERROR-IPC error-IPC))|))
⊨ (outs ← (mbind S(abort$_{lift}$ exec-action$_{id}$-Mon));
P (ERROR-IPC error-IPC# outs)))))))
proof (*cases mbind$_{FailSave}$ S (abort$_{lift}$ exec-action$_{id}$-Mon) σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind$_{FailSave}$ S (abort$_{lift}$ exec-action$_{id}$-Mon) σ = Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a = (aa , b)*
then show *?thesis*
using *hyp0 hyp1*
proof (*cases exec-action$_{id}$-Mon (IPC BUF (RECV caller partner msg)) σ*)
case *None*

then show *?thesis*
using *assms hyp0 hyp1*
by(*simp add*: *exec-action$_{id}$-Mon-def valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *exec-action$_{id}$-Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *ba*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind$_{FailSave}$ S* (*abort$_{lift}$ exec-action$_{id}$-Mon*) *ba* = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *BUF-RECV$_{id}$-def valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action$_{id}$-Mon-def*
split: *split-if-asm*)

```
        next
          case (ERROR-IPC error-IPC)
          assume hyp7:error-codes ba = ERROR-IPC error-IPC
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7
          by (auto simp add:  valid-SE-def bind-SE-def exec-action_id-Mon-def
                split: split-if-asm )
        qed
      qed
    qed
  next
    case (ERROR-MEM error-memory)
    assume hyp4:ac = ERROR-MEM error-memory
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
              (set-error-mem-bufr caller partner σ ba error-memory msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
      assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                 (set-error-mem-bufr caller partner σ ba error-memory msg)
= Some ad
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
      proof (cases ad)
        fix ae bb
        assume hyp6: ad = (ae, bb)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
        by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                          BUF-RECV_id-def
           split        : errors.split option.split list.split-asm split-if-asm)
      qed
    qed
  next
    case (ERROR-IPC error-IPC)
    assume hyp4:ac = ERROR-IPC error-IPC
    then show ?thesis
    using  hyp0 hyp1 hyp2 hyp3 hyp4
    proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
              (set-error-ipc-bufr caller partner σ ba error-IPC msg))
      case None
      then show ?thesis
      by simp
    next
      case (Some ad)
```

```
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-ipc-bufr caller partner σ ba error-IPC msg) = Some
ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              BUF-RECV_id-def
               split        : errors.split option.split list.split-asm split-if-asm)
          qed
        qed
      qed
    qed
  qed
 qed
qed


lemma abort-buf-recv-obvious11:
  (σ ⊨ (outs ← (mbind ((IPC BUF (RECV caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                   P (get-caller-error caller σ # outs)))) ∧
    (caller ∉ dom (( th-flag)σ) ⟶
     (∀ a b.
     (a = NO-ERRORS ⟶ IPC-buf-check-st_id caller partner σ ⟶
      ((σ(|current-thread := caller,
          resource     := foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
                                (zip (get-th-addrs caller σ) (get-msg-values msg σ)),
          thread-list  := update-th-ready caller
                          (update-th-ready partner
                          (thread-list σ)),
          error-codes := NO-ERRORS,
          th-flag      := th-flag σ|))
        ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))) ∧

      (a = NO-ERRORS ∧ msg = [] ⟶ IPC-buf-check-st_id caller partner σ ⟶
       ((σ(|current-thread := caller,
          resource     := resource σ,
          thread-list  := update-th-ready caller
                          (update-th-ready partner
                          (thread-list σ)),
          error-codes := NO-ERRORS,
```

$th\text{-}flag \quad := th\text{-}flag\ \sigma\rrbracket)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))) \wedge$

$(\forall\, error\text{-}IPC.\ a = ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV \longrightarrow$
$\neg IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \longrightarrow$
$((\sigma\llparenthesis current\text{-}thread := caller,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV))\rrbracket)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ \#\ outs)))))))$
by (*simp add*: $abort\text{-}buf\text{-}recv\text{-}obvious10'\ exec\text{-}action_{id}\text{-}Mon\text{-}buf\text{-}recv\text{-}obvious3$, *auto*)

M.4 Symbolic Execution Rules for MAP stage

lemma $abort\text{-}map\text{-}send\text{-}obvious10$:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ (\ (th\text{-}flag\ \sigma))$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$(error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$
$|\ None \Rightarrow (\sigma \models (P\ [])))))$
proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$

then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*

```
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-mem-maps caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-maps caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                       (set-error-ipc-maps caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-map-send-obvious12:
```

$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog)); P\ outs)) =$
$(if\ caller \in dom\ (\ (th\text{-}flag\ \sigma))$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$else\ (case\ ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (NO\text{-}ERRORS\ \#\ outs))) \wedge$
$(((\ th\text{-}flag)\ \sigma)\ caller = None) \wedge$
$(((\ th\text{-}flag)\ \sigma)\ caller =$
$((\ th\text{-}flag)\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))\ caller) \wedge$
$(th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))$
$\mid Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem)) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem)) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ caller =$
$((\ th\text{-}flag)\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg))\ partner)$
$\mid Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$
$((set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog)); P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC)) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC)) \wedge$
$(((\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ caller =$
$((\ th\text{-}flag)\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg))\ partner)$
$\mid None \Rightarrow (\sigma \models (P\ [])))))$

proof $(cases\ mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma)$
case *None*
then show *?thesis*
by *simp*
next
case $(Some\ a)$
assume *hyp0*: $mbind_{FailSave}\ S\ (abort_{lift}\ ioprog)\ \sigma = Some\ a$
then show *?thesis*

using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: ac = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae,\ bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:ac = *ERROR-MEM error-memory*
then show *?thesis*

using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed

qed
qed

lemma *abort-map-send-obvious10''*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
((*caller* ∈ *dom* ((*th-flag*)*σ*) ⟶
(*σ* ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ* # *outs*))))
∧
(*caller* ∉ *dom* ((*th-flag*)*σ*) ⟶
(*ioprog* (*IPC MAP* (*SEND caller partner msg*)) *σ* = *None* ⟶ (*σ* ⊨ (*P* []))) ∧
((∀ *a σ'*.
(*a* = *NO-ERRORS* ⟶ *ioprog* (*IPC MAP* (*SEND caller partner msg*)) *σ* = *Some* (*NO-ERRORS*, *σ'*) ⟶
((*error-tab-transfer caller σ σ'*) ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-memory*. *a* = *ERROR-MEM error-memory* ⟶
ioprog (*IPC MAP* (*SEND caller partner msg*)) *σ* = *Some* (*ERROR-MEM error-memory*, *σ'*) ⟶
((*set-error-mem-maps caller partner σ σ' error-memory msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) ∧
(∀ *error-IPC*. *a* = *ERROR-IPC error-IPC* ⟶
ioprog (*IPC MAP* (*SEND caller partner msg*)) *σ* = *Some* (*ERROR-IPC error-IPC*, *σ'*) ⟶
((*set-error-ipc-maps caller partner σ σ' error-IPC msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ* = *Some a*
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa* , *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) *σ*)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next

case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) (*set-error-mem-maps caller partner* σ *ba error-memory msg*) = *Some ad*
then show *?thesis*

```
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      next
        case (ERROR-IPC error-IPC)
        assume hyp4:ac = ERROR-IPC error-IPC
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4
        proof (cases mbind_FailSave S (abort_lift ioprog)
                    (set-error-ipc-maps caller partner σ ba error-IPC msg))
          case None
          then show ?thesis
          by simp
        next
          case (Some ad)
          assume hyp5: mbind_FailSave S (abort_lift ioprog)
                         (set-error-ipc-maps caller partner σ ba error-IPC msg) =
Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
  qed
qed

lemma abort-map-send-obvious10':
  (σ ⊨ (outs ← (mbind ((IPC MAP (SEND caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                  P (get-caller-error caller σ # outs)))) ∧
    (caller ∉ dom (( th-flag)σ) ⟶
     (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC MAP (SEND caller
partner msg)) σ =
```

```
                                Some (NO-ERRORS, b) ⟶
        ((σ(|current-thread := caller,
            resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                                    (zip msg (get-th-addrs partner σ)),
            thread-list    := update-th-ready caller
                              (update-th-ready partner
                              (thread-list σ)),
            error-codes    := NO-ERRORS,
            th-flag        := th-flag σ|))
          ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS # outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC MAP (SEND caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: exec-action_id-Mon (IPC MAP (SEND caller partner msg)) σ
= Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) ba)
            case None
            then show ?thesis
```

by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*) *ba* = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *MAP-SEND*$_{id}$*-def valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$*-Mon-def*
split: *split-if-asm*)
qed
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$*-Mon*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)

```
          assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-mem-maps caller partner σ ba error-memory msg)
= Some ad
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              MAP-SEND_id-def
               split        : errors.split option.split list.split-asm split-if-asm)
            qed
          qed
        next
          case (ERROR-IPC error-IPC)
          assume hyp4:ac = ERROR-IPC error-IPC
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
          proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                       (set-error-ipc-maps caller partner σ ba error-IPC msg))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
            assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                         (set-error-ipc-maps caller partner σ ba error-IPC msg) =
Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
            proof (cases ad)
              fix ae bb
              assume hyp6: ad = (ae, bb)
              then show ?thesis
              using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
              by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                                MAP-SEND_id-def
                 split        : errors.split option.split list.split-asm split-if-asm)
              qed
            qed
          qed
      qed
    qed
  qed
qed

lemma abort-map-send-obvious11:
```

$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)) =$
$((caller \in dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))) \wedge$

$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\ a\ b.\ (a = NO\text{-}ERRORS \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie dst)))\ (resource\ \sigma)$
$(zip\ msg\ (get\text{-}th\text{-}addrs\ partner\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#$
$outs))))))\ \wedge$

$(caller \notin dom\ ((\ th\text{-}flag)\sigma) \longrightarrow$
$(\forall\ a\ b.\ (a = NO\text{-}ERRORS \wedge msg = [] \longrightarrow$
$((\sigma(\!|current\text{-}thread := caller,$
$resource := (resource\ \sigma)\ ,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!))$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#$
$outs)))))))$
by $(auto\ simp\ add{:}\ abort\text{-}map\text{-}send\text{-}obvious10'\ exec\text{-}action_{id}\text{-}Mon\text{-}map\text{-}send\text{-}obvious3)$

lemma *abort-map-recv-obvious10*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}$
$ioprog));\ P\ outs)) =$
$(if\ caller \in dom\ (\ (th\text{-}flag\ \sigma))$
$then\ (\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#$
$outs)))$
$else\ (case\ ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma\ of$
$Some(NO\text{-}ERRORS, \sigma') \Rightarrow$
$(error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))$
$|\ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Rightarrow$
$((set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem$
$\#\ outs)))$
$|\ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Rightarrow$

```
          ((set-error-ipc-mapr caller partner σ σ' error-IPC msg)
          ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))
        | None ⇒ (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC MAP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC MAP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
         proof (cases mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer caller
σ ba))
            case None
            then show ?thesis
            by simp
          next
            case (Some ad)
             assume hyp5: mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer
caller σ ba) = Some ad
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
```

```
        proof (cases ad)
          fix ae bb
          assume hyp6: ad = (ae, bb)
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
          by(simp add: valid-SE-def bind-SE-def)
        qed
      qed
    next
      case (ERROR-MEM error-memory)
      assume hyp4:ac = ERROR-MEM error-memory
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4
      proof (cases mbind_FailSave S (abort_lift ioprog)
                (set-error-mem-maps caller partner σ ba error-memory msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift ioprog)
                  (set-error-mem-maps caller partner σ ba error-memory msg)
= Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
          fix ae bb
          assume hyp6: ad = (ae, bb)
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
          by(simp add: valid-SE-def bind-SE-def)
        qed
      qed
    next
      case (ERROR-IPC error-IPC)
      assume hyp4:ac = ERROR-IPC error-IPC
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4
      proof (cases mbind_FailSave S (abort_lift ioprog)
                (set-error-ipc-maps caller partner σ ba error-IPC msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift ioprog)
                  (set-error-ipc-maps caller partner σ ba error-IPC msg) =
Some ad
        then show ?thesis
```

```
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
          proof (cases ad)
            fix ae bb
            assume hyp6: ad = (ae, bb)
            then show ?thesis
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(simp add: valid-SE-def bind-SE-def)
          qed
        qed
      qed
    qed
   qed
  qed
qed

lemma abort-map-recv-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC MAP (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
    (if caller ∈ dom ( (th-flag σ))
     then (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ
# outs)))
     else (case ioprog (IPC MAP (RECV caller partner msg)) σ of
            Some(NO-ERRORS, σ') ⇒
            (((error-tab-transfer caller σ σ') ⊨
              (outs ← (mbind S(abort_lift  ioprog)); P (NO-ERRORS # outs)))∧
             ((( th-flag) σ) caller = None) ∧
             ((( th-flag) σ) caller =
              (( th-flag) (error-tab-transfer caller σ σ')) caller) ∧
               (th-flag σ = th-flag (error-tab-transfer caller σ σ')))
          | Some(ERROR-MEM error-mem, σ')⇒
       (((set-error-mem-mapr caller partner σ σ' error-mem msg)
         ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem #
outs)))∧
        ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
           Some (ERROR-MEM error-mem))∧
           ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg))
partner =
           Some (ERROR-MEM error-mem)) ∧
        ((( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg)) caller
=
           (( th-flag) (set-error-mem-maps caller partner σ σ' error-mem msg))
partner))
          | Some(ERROR-IPC error-IPC, σ')⇒
       (((set-error-ipc-mapr caller partner σ σ' error-IPC msg)
         ⊨ (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC#
outs)))∧
       ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
          Some (ERROR-IPC error-IPC))∧
```

```
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner
=
         Some (ERROR-IPC error-IPC)) ∧
      ((( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) caller =
       (( th-flag) (set-error-ipc-maps caller partner σ σ' error-IPC msg)) partner))
         | None ⇒ (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases ioprog (IPC MAP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by (simp add: valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: ioprog (IPC MAP (RECV caller partner msg)) σ = Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3:ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        proof (cases ac)
          case NO-ERRORS
          assume hyp4: ac = NO-ERRORS
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4
         proof (cases mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer caller
σ ba))
           case None
           then show ?thesis
           by simp
          next
           case (Some ad)
            assume hyp5: mbind_FailSave S (abort_lift ioprog)  (error-tab-transfer
caller σ ba) = Some ad
```

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) =

Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*auto simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed
qed
qed
qed

lemma *abort-map-recv-obvious10''*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*)); *P outs*)) =
((*caller* ∈ *dom* ((*th-flag*)*σ*) ⟶
(*σ* ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ* # *outs*))))
∧
(*caller* ∉ *dom* ((*th-flag*)*σ*) ⟶
(*ioprog* (*IPC MAP* (*RECV caller partner msg*)) *σ* = *None* ⟶ (*σ* ⊨ (*P* []))) ∧
((∀ *a σ'*.
(*a* = *NO-ERRORS* ⟶ *ioprog* (*IPC MAP* (*RECV caller partner msg*)) *σ* = *Some* (*NO-ERRORS*, *σ'*) ⟶
((*error-tab-transfer caller σ σ'*) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*));*P* (*NO-ERRORS* # *outs*)))) ∧
(∀ *error-memory*. *a* = *ERROR-MEM error-memory* ⟶
ioprog (*IPC MAP* (*RECV caller partner msg*)) *σ* = *Some* (*ERROR-MEM error-memory*, *σ'*) ⟶
((*set-error-mem-maps caller partner σ σ' error-memory msg*) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-memory* # *outs*)))) ∧
(∀ *error-IPC*. *a* = *ERROR-IPC error-IPC* ⟶
ioprog (*IPC MAP* (*RECV caller partner msg*)) *σ* = *Some* (*ERROR-IPC error-IPC*, *σ'*) ⟶
((*set-error-ipc-maps caller partner σ σ' error-IPC msg*) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))))))))
proof (*cases mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: *mbind*$_{FailSave}$ *S* (*abort*$_{lift}$ *ioprog*) *σ* = *Some a*

then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: $a = (aa\ ,\ b)$
then show *?thesis*
using *hyp0 hyp1*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case *None*
then show *?thesis*
using *assms hyp0 hyp1*
by (*simp add*: *valid-SE-def bind-SE-def*)
next
case (*Some ab*)
assume *hyp2*: *ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*:$ab = (ac,\ ba)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) (*error-tab-transfer caller* σ *ba*) = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: $ad = (ae,\ bb)$
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-MEM error-memory*)
assume *hyp4*:*ac* = *ERROR-MEM error-memory*

then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-mem-maps caller partner* σ *ba error-memory msg*)
= *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
next
case (*ERROR-IPC error-IPC*)
assume *hyp4*:*ac* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)
(*set-error-ipc-maps caller partner* σ *ba error-IPC msg*) =
Some ad
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
by(*simp add*: *valid-SE-def bind-SE-def*)
qed
qed
qed
qed

```
    qed
  qed
qed

lemma abort-map-recv-obvious10':
  (σ ⊨ (outs ← (mbind ((IPC MAP (RECV caller partner msg))#S)
                      (abort_lift  exec-action_id-Mon)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
      (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                   P (get-caller-error caller σ # outs)))) ∧
     (caller ∉ dom (( th-flag)σ) ⟶
      (∀ a b. (a = NO-ERRORS ⟶ exec-action_id-Mon (IPC MAP (RECV caller
partner msg)) σ =
                                Some (NO-ERRORS, b) ⟶
       ((σ⦇current-thread := caller,
          resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                                  (zip msg (get-th-addrs caller σ)),
          thread-list    := update-th-ready caller
                            (update-th-ready partner
                            (thread-list σ)),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ⦈)
         ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))))))
proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon) σ)
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift exec-action_id-Mon) σ = Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa , b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases exec-action_id-Mon (IPC MAP (RECV caller partner msg)) σ)
      case None
      then show ?thesis
      using assms hyp0 hyp1
      by(simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def)
    next
      case (Some ab)
      assume hyp2: exec-action_id-Mon (IPC MAP (RECV caller partner msg)) σ
= Some ab
      then show ?thesis
      using  hyp0 hyp1 hyp2
```

proof (*cases ab*)
fix *ac ba*
assume *hyp3*:*ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
proof (*cases ac*)
case *NO-ERRORS*
assume *hyp4*: *ac* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$-*Mon*) *ba*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ad*)
assume *hyp5*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *exec-action*$_{id}$-*Mon*) *ba* = *Some ad*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5*
proof (*cases ad*)
fix *ae bb*
assume *hyp6*: *ad* = (*ae*, *bb*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6*
proof (*cases error-codes ba*)
case *NO-ERRORS*
assume *hyp7*:*error-codes ba* = *NO-ERRORS*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *MAP-RECV*$_{id}$-*def valid-SE-def bind-SE-def exec-action*$_{id}$-*Mon-def*
split: *split-if-asm option.split-asm*)
next
case (*ERROR-MEM error-memory*)
assume *hyp7*:*error-codes ba* = *ERROR-MEM error-memory*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$-*Mon-def*
split: *split-if-asm*)
next
case (*ERROR-IPC error-IPC*)
assume *hyp7*:*error-codes ba* = *ERROR-IPC error-IPC*
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6 hyp7*
by (*auto simp add*: *valid-SE-def bind-SE-def exec-action*$_{id}$-*Mon-def*
split: *split-if-asm*)
qed
qed

```
      qed
    next
      case (ERROR-MEM error-memory)
      assume hyp4:ac = ERROR-MEM error-memory
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4
      proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                (set-error-mem-maps caller partner σ ba error-memory msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                  (set-error-mem-maps caller partner σ ba error-memory msg)
= Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
          fix ae bb
          assume hyp6: ad = (ae, bb)
          then show ?thesis
          using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
          by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                            MAP-RECV_id-def
             split        : errors.split option.split list.split-asm split-if-asm)
        qed
      qed
    next
      case (ERROR-IPC error-IPC)
      assume hyp4:ac = ERROR-IPC error-IPC
      then show ?thesis
      using  hyp0 hyp1 hyp2 hyp3 hyp4
      proof (cases mbind_FailSave S (abort_lift exec-action_id-Mon)
                (set-error-ipc-maps caller partner σ ba error-IPC msg))
        case None
        then show ?thesis
        by simp
      next
        case (Some ad)
        assume hyp5: mbind_FailSave S (abort_lift exec-action_id-Mon)
                    (set-error-ipc-maps caller partner σ ba error-IPC msg) =
Some ad
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5
        proof (cases ad)
          fix ae bb
          assume hyp6: ad = (ae, bb)
          then show ?thesis
```

```
            using  hyp0 hyp1 hyp2 hyp3 hyp4 hyp5 hyp6
            by(auto simp add: exec-action_id-Mon-def valid-SE-def bind-SE-def
                              MAP-RECV_id-def
               split        : errors.split option.split list.split-asm split-if-asm)
          qed
        qed
      qed
    qed
  qed
 qed
qed

lemma abort-map-recv-obvious11:
 (σ ⊨ (outs ← (mbind ((IPC MAP (RECV caller partner msg))#S)
                     (abort_lift  exec-action_id-Mon)); P outs)) =
  ((caller ∈ dom (( th-flag)σ) ⟶
    (σ ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon));
                 P (get-caller-error caller σ # outs)))) ∧

   (caller ∉ dom (( th-flag)σ) ⟶
    (∀ a b.
     (a = NO-ERRORS ⟶
      ((σ(|current-thread := caller,
          resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                                  (zip msg (get-th-addrs caller σ)),
          thread-list    := update-th-ready caller
                            (update-th-ready partner
                            (thread-list σ)),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ|))
        ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))))) ∧

   (caller ∉ dom (( th-flag)σ) ∧ msg = [] ⟶
    (∀ a b.
     (a = NO-ERRORS ⟶
      ((σ(|current-thread := caller,
          resource       := resource σ,
          thread-list    := update-th-ready caller
                            (update-th-ready partner
                            (thread-list σ)),
          error-codes    := NO-ERRORS,
          th-flag        := th-flag σ|))
        ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
outs)))))))
 by (auto simp add:abort-map-recv-obvious10' exec-action_id-Mon-map-recv-obvious3)
```

M.5 Symbolic Execution Rules for DONE stage

lemma *abort-done-send-obvious11*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)($abort_{lift}$ *ioprog*)); *P outs*)) =
(*if caller* ∈ *dom* ((*th-flag*)*σ*)
then ((*remove-caller-error caller σ*) ⊨
(*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ* # *outs*)))
else (*if ioprog* (*IPC DONE* (*SEND caller partner msg*)) *σ* ≠ *None*
then (*σ* ⊨ (*outs* ← (*mbind S*($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))
else (*σ* ⊨ (*P* []))))
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller σ*))
case *None*
then show *?thesis*
by *simp*
next
case (*Some a*)
assume *hyp0*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*)(*remove-caller-error caller σ*) =
Some a
then show *?thesis*
using *hyp0*
proof (*cases a*)
fix *aa b*
assume *hyp1*: *a* = (*aa*, *b*)
then show *?thesis*
using *hyp0 hyp1*
proof (*cases* $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ*)
case *None*
then show *?thesis*
by *simp*
next
case (*Some ab*)
assume *hyp2*: $mbind_{FailSave}$ *S* ($abort_{lift}$ *ioprog*) *σ* = *Some ab*
then show *?thesis*
using *hyp0 hyp1 hyp2*
proof (*cases ab*)
fix *ac ba*
assume *hyp3*: *ab* = (*ac*, *ba*)
then show *?thesis*
using *hyp0 hyp1 hyp2 hyp3*
by (*auto simp add*: *valid-SE-def bind-SE-def split*: *option.split*)
qed
qed
qed
qed

lemma *abort-done-send-obvious12*:

```
   (σ ⊨ (outs ← (mbind ((IPC DONE (SEND caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
    (if caller ∈ dom (( th-flag)σ)
     then  ((((remove-caller-error caller σ) ⊨
               (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs))) ∧
            ((( th-flag) (remove-caller-error caller σ)) caller = None)          ∧
              caller ≠ partner ∧
             ((( th-flag) σ) partner =
              (( th-flag) (remove-caller-error caller σ)) partner))              ∨
            (((remove-caller-error caller σ) ⊨
               (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ #
outs))) ∧
            ((( th-flag) (remove-caller-error caller σ)) caller = None)          ∧
               caller = partner ∧
        ((( th-flag)  (remove-caller-error caller σ)) partner = None)))
     else (if ioprog (IPC DONE (SEND caller partner msg)) σ ≠ None
           then (σ ⊨ (outs ← (mbind S(abort_lift  ioprog)); P (NO-ERRORS #
outs)))
          else (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
               Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
```

```
      using hyp0 hyp1 hyp2 hyp3
      by (auto simp add: valid-SE-def bind-SE-def split: option.split)
    qed
  qed
 qed
qed

lemma abort-done-send-obvious11':
  (σ ⊨ (outs ← (mbind ((IPC DONE (SEND caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   ((caller ∈ dom (( th-flag)σ) ⟶
     ((remove-caller-error caller σ) ⊨
      (outs ← (mbind S(abort_lift  ioprog)); P (get-caller-error caller σ # outs))))
∧
    (caller ∉ dom (( th-flag)σ) ∧
     ioprog (IPC DONE (SEND caller partner msg)) σ ≠ None ⟶
     (σ ⊨(outs ← (mbind S(abort_lift  ioprog)); P (NO-ERRORS # outs))))∧
    (caller ∉ dom ( (th-flag σ)) ∧
     ioprog (IPC DONE (SEND caller partner msg)) σ = None ⟶
     (σ ⊨( P[]))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
 case None
 then show ?thesis
 by simp
next
 case (Some a)
 assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
             Some a
 then show ?thesis
 using hyp0
 proof (cases a)
  fix aa b
  assume hyp1: a = (aa, b)
  then show ?thesis
  using hyp0 hyp1
  proof (cases mbind_FailSave S (abort_lift ioprog) σ)
   case None
   then show ?thesis
   by simp
  next
   case (Some ab)
   assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
   then show ?thesis
   using hyp0 hyp1 hyp2
   proof (cases ab)
    fix ac ba
    assume hyp3: ab = (ac, ba)
    then show ?thesis
```

```
      using hyp0 hyp1 hyp2 hyp3
      by (simp add: valid-SE-def bind-SE-def split: option.split)
    qed
  qed
 qed
qed



lemma abort-done-recv-obvious11:
  (σ ⊨ (outs ← (mbind ((IPC DONE (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom ( (th-flag σ))
    then  ((remove-caller-error caller σ) ⊨
            (outs ← (mbind S (abort_lift  ioprog)); P (get-caller-error caller σ #
outs)))
    else
     (if ioprog (IPC DONE (RECV caller partner msg)) σ ≠ None
     then (σ ⊨ (outs ← (mbind S (abort_lift  ioprog)); P (NO-ERRORS # outs)))
      else (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
              Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
```

```
      using hyp0 hyp1 hyp2 hyp3
      by (auto simp add: valid-SE-def bind-SE-def split: option.split)
    qed
  qed
 qed
qed

lemma abort-done-recv-obvious12:
  (σ ⊨ (outs ← (mbind ((IPC DONE (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
   (if caller ∈ dom ( (th-flag σ))
    then  ((((remove-caller-error caller σ) ⊨
             (outs ← (mbind S (abort_lift  ioprog)); P (get-caller-error caller σ #
outs))) ∧
           ((( th-flag) (remove-caller-error caller σ)) caller = None)         ∧
             caller ≠ partner ∧
              ((( th-flag) σ) partner =
              (( th-flag) (remove-caller-error caller σ)) partner))            ∨

          (((remove-caller-error caller σ) ⊨
             (outs ← (mbind S (abort_lift  ioprog)); P (get-caller-error caller σ #
outs))) ∧
           ((( th-flag) (remove-caller-error caller σ)) caller = None)         ∧
              caller = partner ∧
                  ((( th-flag)  (remove-caller-error caller σ)) partner = None)))

    else
      (if ioprog (IPC DONE (RECV caller partner msg)) σ ≠ None
      then (σ ⊨ (outs ← (mbind S (abort_lift  ioprog)); P (NO-ERRORS # outs)))
       else (σ ⊨ (P [])))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
               Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
```

```
      by simp
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
        using  hyp0 hyp1 hyp2 hyp3
        by (auto simp add: valid-SE-def bind-SE-def split: option.split)
      qed
    qed
  qed
qed

lemma abort-done-recv-obvious11':
   (σ ⊨ (outs ← (mbind ((IPC DONE (RECV caller partner msg))#S)(abort_lift
ioprog)); P outs)) =
    ((caller ∈ dom (( th-flag)σ) ⟶
      ((remove-caller-error caller σ) ⊨
      (outs ← (mbind S (abort_lift  ioprog)); P (get-caller-error caller σ # outs))))
∧
     (caller ∉ dom (( th-flag)σ) ∧
      ioprog (IPC DONE (RECV caller partner msg)) σ ≠ None ⟶
      (σ ⊨(outs ← (mbind S (abort_lift  ioprog)); P (NO-ERRORS # outs)))) ∧
     (caller ∉ dom ( (th-flag σ)) ∧
     ioprog (IPC DONE (RECV caller partner msg)) σ = None ⟶ (σ ⊨ (P []))))
proof (cases mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ))
  case None
  then show ?thesis
  by simp
next
  case (Some a)
  assume hyp0: mbind_FailSave S (abort_lift ioprog)(remove-caller-error caller σ)
=
               Some a
  then show ?thesis
  using hyp0
  proof (cases a)
    fix aa b
    assume hyp1: a = (aa, b)
    then show ?thesis
    using hyp0 hyp1
    proof (cases mbind_FailSave S (abort_lift ioprog) σ)
      case None
      then show ?thesis
      by simp
```

```
    next
      case (Some ab)
      assume hyp2: mbind_FailSave S (abort_lift ioprog) σ = Some ab
      then show ?thesis
      using hyp0 hyp1 hyp2
      proof (cases ab)
        fix ac ba
        assume hyp3: ab = (ac, ba)
        then show ?thesis
        using hyp0 hyp1 hyp2 hyp3
        by (simp add: valid-SE-def bind-SE-def split:option.split)
      qed
    qed
  qed
qed



lemmas trace-normalizer-errors-TestGen =
     abort-prep-send-obvious10 abort-prep-recv-obvious10 abort-wait-send-obvious10

      abort-wait-recv-obvious10 abort-buf-send-obvious10  abort-buf-recv-obvious10
     abort-done-send-obvious11 abort-done-recv-obvious11 valid-SE-def bind-SE-def
      unit-SE-def

lemmas trace-normalizer-errors-exec-conj-imp-TestGen =
     abort-prep-send-obvious10' abort-prep-recv-obvious10' abort-wait-send-obvious10'

     abort-wait-recv-obvious10' abort-buf-send-obvious10'  abort-buf-recv-obvious10'
       abort-done-send-obvious11' abort-done-recv-obvious11'

end

theory IPC-symbolic-exec-intros
imports IPC-symbolic-exec-rewriting
begin
```

N Introduction Rules for Sequence Testing Scheme

N.1 Introduction Rules for PREP stage

```
lemma abort-prep-send-mbind-TestGen-Pure-intro:
  assumes in-err-state:
        caller ∈ dom ( (th-flag σ)) ⟹
         (σ ⊨ (outs ← (mbind (S)(abort_lift  exec-action_id-Mon));
                              P (get-caller-error caller σ # outs)))
  and not-in-err-state1:
      ⋀a b. caller ∉ dom ( (state_id.th-flag σ)) ⟹
        a = NO-ERRORS ⟹
```

$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ b) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$

and *not-in-err-state2*:
$\bigwedge a\ b\ error\text{-}memory.\ caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}MEM\ error\text{-}memory \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ b) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ error\text{-}memory,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))$

and *not-in-err-state3*:
$\bigwedge a\ b\ error\text{-}IPC.\ caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}IPC\ error\text{-}IPC \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ b) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$

shows $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

using *assms*

by (*simp add*: *abort-prep-send-obvious10'*)

lemma *abort-prep-recv-mbind-TestGen-Pure-intro*:

assumes *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$

and *not-in-err-state1*:
$\bigwedge b.\ caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some$

$(NO\text{-}ERRORS,\ b) \implies$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$

and *not-in-err-state2*:
$\bigwedge b\ error\text{-}memory.$
$caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \implies$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma =$
$Some\ (ERROR\text{-}MEM\ error\text{-}memory,\ b) \implies$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}MEM\ error\text{-}memory,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}MEM\ error\text{-}memory),$
$partner \mapsto (ERROR\text{-}MEM\ error\text{-}memory))|\!)$
$\models (outs \leftarrow (mbind\ S\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}MEM\ error\text{-}memory\ \#\ outs)))$

and *not-in-err-state3*:
$\bigwedge b\ error\text{-}IPC.\ caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \implies$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ b) \implies$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!)$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$

shows $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

using *assms*

by (*simp add*: *abort-prep-recv-obvious10'*)

N.2 Introduction rules for WAIT stage

lemma *abort-wait-send-mbind-TestGen-Pure-intro*:

assumes *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \implies$
$\sigma \models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))$

and *not-in-err-state1*:
$\bigwedge a\ b.\ caller \notin dom\ (\ (state_{id}.th\text{-}flag\ \sigma)) \implies$
$a = NO\text{-}ERRORS \implies$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ b) \implies$
$\sigma(\!|current\text{-}thread := caller,$

thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈ ⊨
(*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*))
and *not-in-err-state3*:
⋀*a b error-IPC*. *caller* ∉ *dom* ((*state*$_{id}$*.th-flag* σ)) ⟹
a = *ERROR-IPC error-IPC* ⟹
exec-action$_{id}$*-Mon* (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) ⟹
σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$*.th-flag* := *state*$_{id}$*.th-flag* σ
(*caller* ↦ (*ERROR-IPC error-IPC*),
partner ↦ (*ERROR-IPC error-IPC*))⦈
⊨ (*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*ERROR-IPC error-IPC*# *outs*))
shows σ ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P outs*)
using *assms*
by (*simp add*: *abort-wait-send-obvious10′*)

lemma *abort-wait-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
σ ⊨ (*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*)); *P* (*get-caller-error caller* σ # *outs*))
and *not-in-err-state1*:
⋀*a b*. *caller* ∉ *dom* ((*state*$_{id}$*.th-flag* σ)) ⟹
a = *NO-ERRORS* ⟹
exec-action$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) ⟹
σ⦇*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*⦈ ⊨
(*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*))
and *not-in-err-state2*:
⋀*a b error-IPC*. *caller* ∉ *dom* ((*state*$_{id}$*.th-flag* σ)) ⟹
a = *ERROR-IPC error-IPC* ⟹
exec-action$_{id}$*-Mon* (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) ⟹
σ⦇*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$*.th-flag* := *state*$_{id}$*.th-flag* σ
(*caller* ↦ (*ERROR-IPC error-IPC*),

$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))\!\rangle$
$\models (outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ WAIT\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by (*auto simp*: *abort-wait-recv-obvious10′ in-err-state*)

N.3 Introduction rules rules for BUF stage

lemma *abort-buf-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))$
and *not-in-err-state1*:
$\bigwedge a\ b.\ caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$a = NO\text{-}ERRORS \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (NO\text{-}ERRORS,\ b) \Longrightarrow$
$\sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (addr,val).\ (m\ (addr :=_{\$}\ val)))\ (resource\ \sigma)$
$(zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS|\!) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))$
and *not-in-err-state2*:
$\bigwedge a\ b\ error\text{-}IPC.\ caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$a = ERROR\text{-}IPC\ error\text{-}IPC \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\ (IPC\ BUF\ (SEND\ caller\ partner\ msg))\ \sigma = Some\ (ERROR\text{-}IPC\ error\text{-}IPC,\ b) \Longrightarrow$
$\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC,$
$state_{id}.th\text{-}flag := state_{id}.th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC))|\!) \models$
$(outs \leftarrow (mbind\ (S)(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (ERROR\text{-}IPC\ error\text{-}IPC\ \#\ outs))$
shows $\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs)$
using *assms*
by (*auto simp* : *abort-buf-send-obvious10′*)

lemma *abort-buf-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
σ ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))
and *not-in-err-state1*:
⋀*a b*. *caller* ∉ *dom* ((*th-flag* σ)) ⟹
a = *NO-ERRORS* ⟹
exec-action$_{id}$-*Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) ⟹
σ(|*current-thread* := *caller*,
resource := *foldl* (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*|)⊨
(*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))
and *not-in-err-state2*:
⋀*a b error-IPC*. *caller* ∉ *dom* ((*th-flag* σ)) ⟹
a = *ERROR-IPC error-IPC* ⟹
exec-action$_{id}$-*Mon* (*IPC BUF* (*RECV caller partner msg*)) σ = *Some* (*ERROR-IPC error-IPC*, *b*) ⟹
σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC*,
state$_{id}$.*th-flag* := *state*$_{id}$.*th-flag* σ
(*caller* ↦ (*ERROR-IPC error-IPC*),
partner ↦ (*ERROR-IPC error-IPC*))|)⊨
(*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*ERROR-IPC error-IPC* # *outs*))
shows σ ⊨ (*outs* ← (*mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*)
using *assms*
by (*auto simp*: *abort-buf-recv-obvious10'*)

N.4 Introduction rules for MAP stage

lemma *abort-map-send-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
σ ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))
and *not-in-err-state1*:
⋀*a b*. *caller* ∉ *dom* ((*th-flag* σ)) ⟹
a = *NO-ERRORS* ⟹
exec-action$_{id}$-*Mon* (*IPC MAP* (*SEND caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) ⟹

$\sigma(\!|$*current-thread* := *caller*,
resource := *foldl* (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs partner* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*$|\!)$ $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))

shows $\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*)

using *assms*

by (*auto simp* : *abort-map-send-obvious10'*)

lemma *abort-map-recv-mbind-TestGen-Pure-intro*:

assumes *in-err-state*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))

and *not-in-err-state1*:
$\bigwedge$*a b*. *caller* $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
a = *NO-ERRORS* $\Longrightarrow$
$exec\text{-}action_{id}$-*Mon* (*IPC MAP* (*RECV caller partner msg*)) σ = *Some* (*NO-ERRORS*, *b*) $\Longrightarrow$
$\sigma(\!|$*current-thread* := *caller*,
resource :=*foldl* (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*$|\!)\models$
(*outs* $\leftarrow$ (*mbind* (*S*)($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))

shows $\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*)

using *assms*

by (*auto simp*: *abort-map-recv-obvious10'*)

N.5 Introduction rules for DONE stage

lemma *abort-done-send-mbind-TestGen-Pure-intro*:

assumes *in-err-state*:
(*caller* $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
(*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind* (*S*)($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*)))

and *not-in-err-state1*:

(*caller* ∉ *dom* ((*state*$_{id}$.*th-flag* σ)) ⟹
σ ⊨ (*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*)))
shows σ ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*)
using *assms*
by (*simp add*: *abort-done-send-obvious11 exec-action*$_{id}$-*Mon-def*)

lemma *abort-done-recv-mbind-TestGen-Pure-intro*:
assumes *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
(*remove-caller-error caller* σ) ⊨
(*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*))
and *not-in-err-state1*:
caller ∉ *dom* ((*state*$_{id}$.*th-flag* σ)) ⟹
σ ⊨ (*outs* ← (*mbind* (*S*)(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))
shows σ ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*)
using *assms*
by (*simp add*: *abort-done-recv-obvious11 exec-action*$_{id}$-*Mon-def*)

end

theory *IPC-symbolic-exec-elims*
imports *IPC-symbolic-exec-rewriting IPC-symbolic-exec-intros ../../../../src/TestLib*
begin

O Elimination rules for Symbolic Execution of a Test Specification

lemma *threa-table-obvious*:
(*caller* ∉ *dom* ((*th-flag* σ))) = ((*th-flag* σ) *caller* = *None*)
by *auto*

lemma *threa-table-obvious'*:
((*th-flag* σ) *caller* = *None*) = (*caller* ∉ *dom* ((*th-flag* σ)))
by *auto*

O.1 Symbolic Execution rules for PREP SEND

HOL representation

lemma *abort-prep-send-mbindFSave-E*:
assumes *valid-exec*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$

ioprog));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
(σ ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀σ′.
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*,
σ′) ⟹
((*error-tab-transfer caller* σ σ′)
⊨ (*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))
⟹*Q*
and *not-in-err-state-Some2*:
⋀σ′ *error-mem*.
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM*
error-mem, σ′) ⟹
((*set-error-mem-waitr caller partner* σ σ′ *error-mem msg*) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* #
outs))) ⟹*Q*
and *not-in-err-state-Some3*:
⋀σ′ *error-IPC*.
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC*
error-IPC, σ′) ⟹
((*set-error-ipc-waitr caller partner* σ σ′ *error-IPC msg*) ⊨
(*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*)))
⟹*Q*
and *not-in-err-state-None*:
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC PREP* (*SEND caller partner msg*)) σ = *None* ⟹
(σ ⊨ (*P* [])) ⟹ *Q*
shows *Q*
proof (*cases caller* ∈ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *case-tac a*, *simp*,

simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-send-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC PREP* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$exec\text{-}action_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$exec\text{-}action_{id}$-*Mon-prep-fact1 caller partner* σ $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*)))
$\Longrightarrow$ *Q*
and
not-in-err-exec2:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$\neg exec\text{-}action_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
(σ(|*current-thread* := *caller*,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-MEM not-valid-sender-addr-in-PREP-SEND*,
$state_{id}$.*th-flag* := *th-flag* σ
(*caller* $\mapsto$ (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*),
partner $\mapsto$ (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND*))|) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*ERROR-MEM not-valid-sender-addr-in-PREP-SEND* # *outs*)))$\Longrightarrow$
Q
and
not-in-err-exec31:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$exec\text{-}action_{id}$-*Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*)$\Longrightarrow$
IPC-params-c2 ((*the o thread-list* σ) *partner*)$\Longrightarrow$
$\neg$ *IPC-params-c6 caller* ((*the o thread-list* σ) *partner*) $\Longrightarrow$

$(\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{thread\text{-}list} := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},$
$\mathit{error\text{-}codes} := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND},$
$\mathit{th\text{-}flag} := \mathit{th\text{-}flag}\ \sigma$
$(\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND}),$
$\mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND}))|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}SEND}\#\ \mathit{outs})))\Longrightarrow Q$

and

$\mathit{not\text{-}in\text{-}err\text{-}exec32}:$
$\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon\text{-}prep\text{-}fact0\ caller\ partner}\ \sigma\ \mathit{msg} \Longrightarrow$
$\neg \mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow$
$\neg \mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow$
$(\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{thread\text{-}list} := \mathit{update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma)},$
$\mathit{error\text{-}codes} := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND},$
$\mathit{th\text{-}flag} := \mathit{th\text{-}flag}\ \sigma$
$(\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND}),$
$\mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND}))|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND}\#\ \mathit{outs})))\Longrightarrow Q$

and

$\mathit{not\text{-}in\text{-}err\text{-}exec33}:$
$\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon\text{-}prep\text{-}fact0\ caller\ partner}\ \sigma\ \mathit{msg} \Longrightarrow$
$\neg \mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow$
$\mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow$
$\mathit{IPC\text{-}params\text{-}c6\ caller}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \mathit{partner}) \Longrightarrow$
$(\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{thread\text{-}list} := \mathit{update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma)},$
$\mathit{error\text{-}codes} := \mathit{NO\text{-}ERRORS}|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs})))\Longrightarrow Q$

shows Q

apply (*insert valid-exec*)
apply (*elim abort-prep-send-mbindFSave-E*)
apply (*simp add*: *in-err-exec*)
apply (*simp add*: $\mathit{exec\text{-}action}_{id}$*-Mon-prep-send-obvious3*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: $\mathit{exec\text{-}action}_{id}$*-Mon-prep-send-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)

apply (*simp add*: *not-in-err-exec2 exec-action$_{id}$-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action$_{id}$-Mon-prep-send-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

O.2 Symbolic Execution rules for PREP RECV

lemma *abort-prep-recv-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs)))$

$\Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by $(subst\ (asm)\ abort\text{-}prep\text{-}recv\text{-}obvious10,\ elim\ in\text{-}err\text{-}state,\ simp)$
next
case *False*
then show *?thesis*
using *valid-exec*
proof $(cases\ ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma)$
case $(Some\ a)$
then show *?thesis*
using *valid-exec False*
by $(subst\ (asm)\ abort\text{-}prep\text{-}recv\text{-}obvious10,\ simp,\ case\text{-}tac\ a,\ simp,$
$simp\ split: errors.split\text{-}asm,\ elim\ not\text{-}in\text{-}err\text{-}state\text{-}Some1,$
$auto\ intro: not\text{-}in\text{-}err\text{-}state\text{-}Some2\ not\text{-}in\text{-}err\text{-}state\text{-}Some3)$
next
case *None*
then show *?thesis*
using *valid-exec False*
by $(subst\ (asm)\ abort\text{-}prep\text{-}recv\text{-}obvious10,\ simp,\ elim\ not\text{-}in\text{-}err\text{-}state\text{-}None)$
qed
qed

lemma *abort-prep-recv-HOL-elim21*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:

$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$

$\Longrightarrow Q$
and
not-in-err-exec2:

$$\begin{array}{l}
\quad \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \neg \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon\text{-}prep\text{-}fact0}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \mathit{msg} \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}MEM}\ \mathit{not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV}, \\
\qquad \mathit{state}_{id}.\mathit{th\text{-}flag} := \mathit{th\text{-}flag}\ \sigma \\
\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}MEM}\ \mathit{not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV}), \\
\qquad \mathit{partner} \mapsto (\mathit{ERROR\text{-}MEM}\ \mathit{not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\qquad P\ (\mathit{ERROR\text{-}MEM}\ \mathit{not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV}\ \#\ \mathit{outs})))\Longrightarrow Q
\end{array}$$

and
not-in-err-exec31:

$$\begin{array}{l}
\quad \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon\text{-}prep\text{-}fact0}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \mathit{msg} \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow \\
\quad \mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow \\
\quad \neg\ \mathit{IPC\text{-}params\text{-}c6}\ \mathit{caller}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \mathit{partner}) \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} := \mathit{th\text{-}flag}\ \sigma \\
\qquad\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV}), \\
\qquad\qquad \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\qquad P\ (\ \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV}\#\ \mathit{outs})))\Longrightarrow Q
\end{array}$$

and
not-in-err-exec32:

$$\begin{array}{l}
\quad \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon\text{-}prep\text{-}fact0}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \mathit{msg} \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}params\text{-}c1}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}params\text{-}c2}\ ((\mathit{the}\ o\ \mathit{thread\text{-}list}\ \sigma)\ \ \mathit{partner}) \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} := \mathit{th\text{-}flag}\ \sigma \\
\qquad\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV}), \\
\qquad\qquad \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\qquad P\ (\ \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV}\#\ \mathit{outs})))\Longrightarrow Q
\end{array}$$

and
not-in-err-exec33:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
exec-action$_{id}$*-Mon-prep-fact0 caller partner* σ *msg* $\Longrightarrow$
$\neg$*IPC-params-c1* ((*the o thread-list* σ) *partner*) $\Longrightarrow$
IPC-params-c2 ((*the o thread-list* σ) *partner*) $\Longrightarrow$
IPC-params-c6 caller ((*the o thread-list* σ) *partner*)$\Longrightarrow$
(σ($\|$*current-thread* := *caller*,
thread-list := *update-th-ready caller* (*thread-list* σ),
error-codes := *NO-ERRORS*$\|$) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$*-Mon*));*P* (*NO-ERRORS* # *outs*)))$\Longrightarrow$*Q*

shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-prep-recv-mbindFSave-E*)
apply (*simp add*: *in-err-exec*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-recv-obvious3*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-recv-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec2 exec-action*$_{id}$*-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-prep-recv-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

O.3 Symbolic Execution rules for WAIT SEND

lemma *abort-wait-send-mbindFSave-E*:
assumes *valid-exec*:
(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:

$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by $(subst\ (asm)\ abort\text{-}wait\text{-}send\text{-}obvious10,\ elim\ in\text{-}err\text{-}state,\ simp)$
next
case *False*
then show *?thesis*
using *valid-exec*
proof $(cases\ ioprog\ (IPC\ WAIT\ (SEND\ caller\ partner\ msg))\ \sigma)$
case $(Some\ a)$
then show *?thesis*
using *valid-exec False*
by $(subst\ (asm)\ abort\text{-}wait\text{-}send\text{-}obvious10,\ simp,\ case\text{-}tac\ a,\ simp,$
$simp\ split: errors.split\text{-}asm,\ elim\ not\text{-}in\text{-}err\text{-}state\text{-}Some1,$
$auto\ intro: not\text{-}in\text{-}err\text{-}state\text{-}Some2\ not\text{-}in\text{-}err\text{-}state\text{-}Some3)$
next

case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-send-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* *S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
IPC-send-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
IPC-params-c4 caller partner $\Longrightarrow$
IPC-params-c5 partner σ $\Longrightarrow$
(σ($\|$*current-thread* := *caller*,
thread-list := *update-th-waiting caller* (*thread-list* σ),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ)$\|$)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec21:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$\neg$*IPC-send-comm-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$
(σ($\|$*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-1-in-WAIT-SEND*,
th-flag := *th-flag* σ
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-WAIT-SEND*))$\|$) $\models$
(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-1-in-WAIT-SEND*# *outs*)))$\Longrightarrow$ *Q*
and
not-in-err-exec22:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
IPC-send-comm-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
$\neg$*IPC-params-c4 caller partner* $\Longrightarrow$
(σ($\|$*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),

$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma$
$(caller \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND))\rangle\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift} \ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}SEND\#\ outs)))\Longrightarrow Q$

and

not-in-err-exec23:

$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$(thread\text{-}list\ \sigma)\ caller = None \Longrightarrow$
$(\sigma\langle\!|current\text{-}thread := caller\ ,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma$
$(caller \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND))\rangle\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift} \ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}SEND\#\ outs)))\Longrightarrow Q$

and

not-in-err-exec24:

$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$\exists\, th.\ (thread\text{-}list\ \sigma)\ caller = Some\ th \Longrightarrow$
$(\sigma\langle\!|current\text{-}thread := caller\ ,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma$
$(caller \ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND))\rangle\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift} \ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}SEND\#\ outs)))\Longrightarrow Q$

shows Q

apply (*insert valid-exec*)
apply (*elim abort-wait-send-mbindFSave-E*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: $exec\text{-}action_{id}$*-Mon-wait-send-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: $exec\text{-}action_{id}$*-Mon-def* $WAIT\text{-}SEND_{id}$*-def split*: *split-if-asm option.split-asm*)
apply (*simp only*: $exec\text{-}action_{id}$*-Mon-wait-send-obvious4*)
apply (*auto*)
apply (*erule contrapos-np*)

apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*elim not-in-err-exec23*)
apply *simp-all*
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

O.4 Symbolic Execution rules for WAIT RECV

lemma *abort-wait-recv-mbindFSave-E*:
assumes *valid-exec*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag σ*)) ⟹
(*σ* ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ # outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀*σ′*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*RECV caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ′*) ⟹
((*error-tab-transfer caller σ σ′*) ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS # outs*))) ⟹*Q*
and *not-in-err-state-Some2*:
⋀*σ′ error-mem*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*RECV caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ′*) ⟹

((*set-error-mem-waitr caller partner* σ σ' *error-mem msg*) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*))) $\Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*set-error-ipc-waitr caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-recv-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and

not-in-err-exec1:

$$
\begin{array}{l}
\quad \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad \mathit{IPC\text{-}params\text{-}c4}\ \mathit{caller}\ \mathit{partner} \Longrightarrow \\
\quad \mathit{IPC\text{-}params\text{-}c5}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}, \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}waiting}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{NO\text{-}ERRORS}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma|\!) \\
\qquad\quad \models (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift} \quad \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs}))) \Longrightarrow Q
\end{array}
$$

and

not-in-err-exec21:

$$
\begin{array}{l}
\quad \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma \\
\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}), \\
\qquad\ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift} \quad \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\quad P\ (\ \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}\ \#\ \mathit{outs}))) \Longrightarrow Q
\end{array}
$$

and

not-in-err-exec22:

$$
\begin{array}{l}
\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}params\text{-}c4}\ \mathit{caller}\ \mathit{partner} \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma \\
\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}), \\
\qquad\ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift} \quad \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\quad P\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\ \#\ \mathit{outs}))) \Longrightarrow Q
\end{array}
$$

and

not-in-err-exec23:

$$
\begin{array}{l}
\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\quad \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad \mathit{IPC\text{-}params\text{-}c4}\ \mathit{caller}\ \mathit{partner} \Longrightarrow \\
\quad \neg \mathit{IPC\text{-}params\text{-}c5}\ \mathit{partner}\ \sigma \Longrightarrow \\
\quad (\mathit{thread\text{-}list}\ \sigma)\ \mathit{caller} = \mathit{None} \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma
\end{array}
$$

$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV))\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV\#\ outs))) \Longrightarrow Q$

and

not-in-err-exec24:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}recv\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$\neg IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$\exists\, th.\ (thread\text{-}list\ \sigma)\ caller = Some\ th \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV))\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}5\text{-}in\text{-}WAIT\text{-}RECV\#\ outs))) \Longrightarrow Q$

shows Q
apply (*insert valid-exec*)
apply (*elim abort-wait-recv-mbindFSave-E*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: $exec\text{-}action_{id}$*-Mon-wait-recv-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: $exec\text{-}action_{id}$*-Mon-def* $WAIT\text{-}RECV_{id}$*-def split*: *split-if-asm option.split-asm*)
apply (*simp only*: $exec\text{-}action_{id}$*-Mon-wait-recv-obvious4*)
apply (*auto*)
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*elim not-in-err-exec23*)
apply *simp-all*
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious′*)

apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action*$_{id}$*-Mon-def*)
done

O.5 Symbolic Execution rules for BUF SEND

lemma *abort-buf-send-mbindFSave-E*:
assumes *valid-exec*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag σ*)) ⟹
(*σ* ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller σ # outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀*σ*′.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ*′)
⟹
((*error-tab-transfer caller σ σ*′) ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS # outs*))) ⟹*Q*
and *not-in-err-state-Some2*:
⋀*σ*′ *error-mem*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ*′) ⟹
((*set-error-mem-bufs caller partner σ σ*′ *error-mem msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem # outs*))) ⟹*Q*
and *not-in-err-state-Some3*:
⋀*σ*′ *error-IPC*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-IPC error-IPC*, *σ*′) ⟹
((*set-error-ipc-bufs caller partner σ σ*′ *error-IPC msg*)
⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) ⟹*Q*
and *not-in-err-state-None*:
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *None* ⟹
(*σ* ⊨ (*P* [])) ⟹ *Q*
shows *Q*
proof (*cases caller* ∈ *dom* ((*th-flag σ*)))
case *True*
then show *?thesis*

using *valid-exec*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-buf-send-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* *S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
IPC-buf-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
(σ(|*current-thread* := *caller*,
resource := *foldl* (λ*m* (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)
(*zip* (*get-th-addrs partner* σ) (*get-msg-values msg* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* σ|)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec12:
caller $\notin$ *dom* (*th-flag* σ) $\Longrightarrow$
IPC-buf-check-st$_{id}$ *caller partner* σ $\Longrightarrow$
msg = [] $\Longrightarrow$
(σ(|*current-thread* := *caller*,

$resource \quad := resource\ \sigma,$
$thread\text{-}list \ := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes \ := NO\text{-}ERRORS,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma(\!|)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

and
not-in-err-exec2:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \Longrightarrow$
$(\sigma(|current\text{-}thread := caller\ ,$
$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND))|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\#\ outs))) \Longrightarrow Q$
shows Q
apply(*insert valid-exec*)
apply (*subst* (*asm*) *abort-buf-send-obvious11*)
using *in-err-exec not-in-err-exec1 not-in-err-exec2*
apply *auto*
done

O.6 Symbolic Execution rules for BUF RECV

lemma *abort-buf-recv-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$

ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ') $\Longrightarrow$
((*set-error-mem-bufr caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem # outs*))) $\Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*set-error-ipc-bufr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC# outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC BUF* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-buf-recv-HOL-elim21*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));

P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ Q

and

not-in-err-exec1:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$

IPC-buf-check-st$_{id}$ *caller partner* σ $\Longrightarrow$

(σ($\!|$*current-thread* := *caller*,

resource := *foldl* (λm (*addr*,*val*). (*m* (*addr*:=$_{\$}$ *val*))) (*resource* σ)

(*zip* (*get-th-addrs caller* σ) (*get-msg-values msg* σ)),

thread-list := *update-th-ready caller*

(*update-th-ready partner*

(*thread-list* σ)),

error-codes := *NO-ERRORS*,

th-flag := *th-flag* σ|)

$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); P (*NO-ERRORS* # *outs*))) $\Longrightarrow$ Q

and

not-in-err-exec2:

caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$

$\neg$ *IPC-buf-check-st*$_{id}$ *caller partner* σ $\Longrightarrow$

(σ($\!|$*current-thread* := *caller* ,

thread-list := *update-th-current caller* (*thread-list* σ),

error-codes := *ERROR-IPC error-IPC-1-in-BUF-RECV*,

th-flag := *th-flag* σ

(*caller* $\mapsto$ (*ERROR-IPC error-IPC-1-in-BUF-RECV*),

partner $\mapsto$ (*ERROR-IPC error-IPC-1-in-BUF-RECV*))|) $\models$

(*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));

P (*ERROR-IPC error-IPC-1-in-BUF-RECV*# *outs*))) $\Longrightarrow$ Q

shows Q

apply(*insert valid-exec*)

apply (*subst* (*asm*) *abort-buf-recv-obvious11*)

using *in-err-exec not-in-err-exec1 not-in-err-exec2*

apply *auto*

done

O.7 Symbolic Execution rules for MAP SEND

lemma *abort-map-send-mbindFSave-E*:

assumes *valid-exec*:

(σ $\models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))

and *in-err-state*:

caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$

(σ $\models$

(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ Q

and *not-in-err-state-Some1*:

$\bigwedge\sigma'$.

(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$

ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*NO-ERRORS*,

σ') $\Longrightarrow$
((*error-tab-transfer caller* σ σ') $\models$
(*outs* $\leftarrow$ (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-Some2*:
$\bigwedge\sigma'$ *error-mem*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-MEM error-mem*, σ') $\Longrightarrow$
((*set-error-mem-maps caller partner* σ σ' *error-mem msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow$*Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-send-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-send-HOL-elim2*:

assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie dst)))\ (resource\ \sigma)$
$(zip\ msg\ (get\text{-}th\text{-}addrs\ partner\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))$
$\Longrightarrow Q$

and
not-in-err-exec12:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow msg = [] \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource := resource\ \sigma,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs)))$
$\Longrightarrow Q$
shows Q
apply(*insert valid-exec*)
apply (*subst* (*asm*) *abort-map-send-obvious11*)
using *in-err-exec not-in-err-exec1 not-in-err-exec12*
apply *auto*
done

O.8 Symbolic Execution rules for MAP RECV

lemma *abort-map-recv-mbindFSave-E*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}$
$ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$

$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (\ (th\text{-}flag\ \sigma)))$
case *True*
then show *?thesis*
using *valid-exec*
by $(subst\ (asm)\ abort\text{-}map\text{-}recv\text{-}obvious10,\ elim\ in\text{-}err\text{-}state,\ simp)$
next
case *False*
then show *?thesis*
proof $(cases\ ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma)$
case $(Some\ a)$
then show *?thesis*
using *valid-exec False Some*
by $(subst\ (asm)\ abort\text{-}map\text{-}recv\text{-}obvious10,$
case-tac a,simp split: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*

using *valid-exec False*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-recv-HOL-elim2*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* *S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
(σ($\|$*current-thread* := *caller*,
resource := *foldl* (λ*m* (*src*,*dst*). (*m* (*src*$\bowtie$ *dst*))) (*resource* σ)
(*zip msg* (*get-th-addrs caller* σ)),
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* $\sigma$$\|$)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*

and
not-in-err-exec12:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$ *msg* = [] $\Longrightarrow$
(σ($\|$*current-thread* := *caller*,
resource := *resource* σ,
thread-list := *update-th-ready caller*
(*update-th-ready partner*
(*thread-list* σ)),
error-codes := *NO-ERRORS*,
th-flag := *th-flag* $\sigma$$\|$)
$\models$ (*outs* $\leftarrow$ (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*
shows *Q*
apply(*insert valid-exec*)
apply (*subst* (*asm*) *abort-map-recv-obvious11*)
using *in-err-exec not-in-err-exec1 not-in-err-exec12*
apply *auto*
done

O.9 Symbolic Execution rules for DONE SEND

lemma *abort-done-send-mbindFSave-E*:

assumes *valid-exec*:
(σ ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)(*abort*$_{lift}$ *ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
((*remove-caller-error caller* σ) ⊨
(*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
⟹ *Q*
and *not-in-err-state-Some*:
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ ≠ *None* ⟹
(σ ⊨ (*outs* ← (*mbind S* (*abort*$_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))
⟹*Q*
and *not-in-err-state-None*:
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC DONE* (*SEND caller partner msg*)) σ = *None* ⟹
(σ ⊨ (*P* [])) ⟹ *Q*
shows *Q*
proof (*cases caller* ∈ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-done-send-obvious11* , *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*SEND caller partner msg*)) σ ≠ *None*)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-send-obvious11*, *simp only*: *False comp-apply*)
next
case *False*
then show *?thesis*
using *assms not-in-err-state-None*
by (*metis* (*mono-tags*) *comp-apply in-err-state False abort-done-send-obvious11*)
qed
qed

lemma *abort-done-send-HOL-elim1*:
assumes
valid-exec: (σ ⊨ (*outs* ← (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)
(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))

and *in-err-exec*:
caller ∈ *dom* ((*th-flag* σ)) ⟹
(((*remove-caller-error caller* σ) ⊨ (*outs* ← (*mbind S*(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));

P (*get-caller-error caller* σ # *outs*))) ⟹ *Q*)

and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* S(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));P (*NO-ERRORS* # *outs*))) $\Longrightarrow$ Q
shows Q
using *assms*
by (*rule abort-done-send-mbindFSave-E*, *simp-all add*: *exec-action*$_{id}$-*Mon-def*)

O.10 Symbolic Execution rules for DONE SEND

lemma *abort-done-recv-mbindFSave-E*:
assumes *valid-exec*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*RECV caller partner msg*))#S)(*abort*$_{lift}$ *ioprog*));P *outs*))
and *in-err-state*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind* S (*abort*$_{lift}$ *ioprog*)); P (*get-caller-error caller* σ # *outs*)))
$\Longrightarrow$ Q
and *not-in-err-state-Some*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None* $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* S (*abort*$_{lift}$ *ioprog*)); P (*NO-ERRORS* # *outs*)))
$\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC DONE* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (P [])) $\Longrightarrow$ Q
shows Q
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-done-recv-obvious11* , *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*RECV caller partner msg*)) $\sigma \neq$ *None*)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-recv-obvious11*, *simp only*: *False*)
next
case *False*
then show *?thesis*
using *assms not-in-err-state-None*
by (*metis* (*mono-tags*) *in-err-state False abort-done-recv-obvious11*)
qed

qed

lemma *abort-done-recv-HOL-elim1*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$

and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$(((remove\text{-}caller\text{-}error\ caller\ \sigma) \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$

$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q)$
and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
shows Q
using *assms*
by (*rule abort-done-recv-mbindFSave-E*, *simp-all add*: $exec\text{-}action_{id}$*-Mon-def*)

P Rules with detailed Constraints

P.1 Symbolic Execution rules for PREP SEND

HOL representation

lemma *abort-prep-send-mbindFSave-E′*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$

$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}MEM\ error\text{-}mem,\ \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$

and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(ERROR\text{-}IPC\ error\text{-}IPC,\ \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}waitr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$

and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$

shows Q

proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC PREP* (*SEND caller partner msg*)) σ)
case (*Some a*)

then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-prep-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-send-HOL-elim21'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma) = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$
$\Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))$
$caller =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND\ msg)))$
$partner =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}sender\text{-}addr\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$

$$\textit{not-valid-sender-addr-in-PREP-SEND msg})))$$

$\textit{caller} =$

$$(\ (\textit{th-flag}\ (\textit{set-error-mem-maps caller partner}\ \sigma\ \sigma\ \textit{not-valid-sender-addr-in-PREP-SEND msg})))$$

$\textit{partner} \Longrightarrow$

$$(\sigma(\!|\textit{current-thread} := \textit{caller},\ \textit{thread-list} := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),\ \textit{error-codes} := \textit{ERROR-MEM not-valid-sender-addr-in-PREP-SEND},\ \textit{th-flag} := \textit{th-flag}\ \sigma\ (\textit{caller} \mapsto (\textit{ERROR-MEM not-valid-sender-addr-in-PREP-SEND}),\ \textit{partner} \mapsto (\textit{ERROR-MEM not-valid-sender-addr-in-PREP-SEND}))|\!) \models (\textit{outs} \leftarrow (\textit{mbind}\ S(\textit{abort}_{lift}\ \ \textit{exec-action}_{id}\textit{-Mon}));\ P\ (\textit{ERROR-MEM not-valid-sender-addr-in-PREP-SEND}\ \#\ \textit{outs})))\Longrightarrow Q$$

and

$\textit{not-in-err-exec31}$:

$$\textit{caller} \notin \textit{dom}\ (\ (\textit{th-flag}\ \sigma)) \Longrightarrow \textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner}\ \sigma\ \textit{msg} \Longrightarrow \neg\textit{IPC-params-c1}\ ((\textit{the o thread-list}\ \sigma)\ \ \textit{partner}) \Longrightarrow \textit{IPC-params-c2}\ ((\textit{the o thread-list}\ \sigma)\ \ \textit{partner}) \Longrightarrow \neg\ \textit{IPC-params-c6 caller}\ ((\textit{the o thread-list}\ \sigma)\ \textit{partner}) \Longrightarrow (\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma\ \textit{error-IPC-22-in-PREP-SEND msg})))\ \textit{caller} = \textit{Some}\ (\textit{ERROR-IPC error-IPC-22-in-PREP-SEND}) \Longrightarrow (\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma\ \textit{error-IPC-22-in-PREP-SEND msg})))\ \textit{partner} = \textit{Some}\ (\textit{ERROR-IPC error-IPC-22-in-PREP-SEND}) \Longrightarrow (\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma\ \textit{error-IPC-22-in-PREP-SEND msg})))\ \textit{caller} = (\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma\ \textit{error-IPC-22-in-PREP-SEND msg})))\ \textit{partner} \Longrightarrow (\sigma(\!|\textit{current-thread} := \textit{caller},\ \textit{thread-list} := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),\ \textit{error-codes} := \textit{ERROR-IPC error-IPC-22-in-PREP-SEND},\ \textit{th-flag} := \textit{th-flag}\ \sigma\ (\textit{caller} \mapsto (\textit{ERROR-IPC error-IPC-22-in-PREP-SEND}),\ \textit{partner} \mapsto (\textit{ERROR-IPC error-IPC-22-in-PREP-SEND}))|\!) \models (\textit{outs} \leftarrow (\textit{mbind}\ S(\textit{abort}_{lift}\ \ \textit{exec-action}_{id}\textit{-Mon}));\ P\ (\ \textit{ERROR-IPC error-IPC-22-in-PREP-SEND}\#\ \textit{outs})))\Longrightarrow Q$$

and

$\textit{not-in-err-exec32}$:

$$\textit{caller} \notin \textit{dom}\ (\ (\textit{th-flag}\ \sigma)) \Longrightarrow \textit{exec-action}_{id}\textit{-Mon-prep-fact0 caller partner}\ \sigma\ \textit{msg} \Longrightarrow \neg\textit{IPC-params-c1}\ ((\textit{the o thread-list}\ \sigma)\ \ \textit{partner}) \Longrightarrow \neg\textit{IPC-params-c2}\ ((\textit{the o thread-list}\ \sigma)\ \ \textit{partner}) \Longrightarrow (\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma\ \textit{error-IPC-23-in-PREP-SEND msg})))\ \textit{caller} =$$

$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND) \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ caller =$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND))|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}SEND\#\ outs))) \Longrightarrow Q$

and

$not\text{-}in\text{-}err\text{-}exec33$:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

shows Q
apply $(insert\ valid\text{-}exec)$
apply $(elim\ abort\text{-}prep\text{-}send\text{-}mbindFSave\text{-}E')$
apply $(simp\ add{:}\ in\text{-}err\text{-}exec)$
apply $(simp\ only{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}obvious3)$
apply $auto$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(rule\ not\text{-}in\text{-}err\text{-}exec1)$
apply $(simp\text{-}all\ add{:}\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}send\text{-}obvious4)$
apply $auto$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(fold\ update\text{-}th\text{-}current.simps)$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$

apply (*simp add*: *not-in-err-exec2 exec-action$_{id}$-Mon-prep-fact0-def*)
apply (*simp add*: *exec-action$_{id}$-Mon-prep-send-obvious5*)
apply *auto*
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec31*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

P.2 Symbolic Execution rules for PREP RECV

lemma *abort-prep-recv-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge \sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ PREP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$

```
       (  (th-flag (set-error-mem-maps caller partner σ σ' error-mem msg))) caller
=
      (  (th-flag (set-error-mem-maps caller partner σ σ' error-mem msg))) partner
⟹
       ((set-error-mem-waitr caller partner σ σ' error-mem msg) ⊨
          (outs ← (mbind S(abort_lift  ioprog)); P (ERROR-MEM error-mem # outs))) ⟹Q
  and not-in-err-state-Some3:
       ⋀σ' error-IPC.
       (caller ∉ dom ( (th-flag σ))) ⟹
        ioprog (IPC PREP (RECV caller partner msg)) σ =  Some(ERROR-IPC error-IPC, σ') ⟹
       (  (th-flag (set-error-ipc-maps caller partner σ σ' error-IPC msg))) caller =

        Some (ERROR-IPC error-IPC) ⟹
       (  (th-flag (set-error-ipc-maps caller partner σ σ' error-IPC msg))) partner
=
        Some (ERROR-IPC error-IPC)⟹
       (  (th-flag (set-error-ipc-maps caller partner σ σ' error-IPC msg))) caller =
       (  (th-flag (set-error-ipc-maps caller partner σ σ' error-IPC msg))) partner
⟹
       ((set-error-ipc-waitr caller partner σ σ' error-IPC msg) ⊨
       (outs ← (mbind S(abort_lift  ioprog)); P ( ERROR-IPC error-IPC# outs)))
⟹Q
  and not-in-err-state-None:
      (caller ∉ dom ( (th-flag σ))) ⟹
       ioprog (IPC PREP (RECV caller partner msg)) σ = None ⟹
       (σ ⊨ (P [])) ⟹ Q
  shows Q
proof (cases caller ∈ dom ( (th-flag σ)))
  case True
  then show ?thesis
  using valid-exec
  by (subst (asm) abort-prep-recv-obvious10, elim in-err-state, simp)
next
  case False
  then show ?thesis
  using valid-exec
  proof (cases ioprog (IPC PREP (RECV caller partner msg)) σ)
    case (Some a)
    then show ?thesis
    using valid-exec False
    by (subst (asm) abort-prep-recv-obvious10, simp, case-tac a, simp,
        simp split: errors.split-asm, elim not-in-err-state-Some1,
        auto intro: not-in-err-state-Some2 not-in-err-state-Some3)
  next
    case None
    then show ?thesis
    using valid-exec False
```

by (*subst* (*asm*) *abort-prep-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-prep-recv-HOL-elim21'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ PREP\ (RECV\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact1\ caller\ partner\ \sigma \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs)))$
$\Longrightarrow Q$
and
not-in-err-exec2:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ msg)))$
$caller =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ msg)))$
$partner =$
$Some\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ msg)))$
$caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ msg)))$
$partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma$

$(caller \mapsto (ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV))\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (ERROR\text{-}MEM\ not\text{-}valid\text{-}receiver\text{-}addr\text{-}in\text{-}PREP\text{-}RECV\ \#\ outs)))\Longrightarrow Q$

and

not-in-err-exec31:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\Longrightarrow$
$\neg\ IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV)\Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV))\!\rangle \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}22\text{-}in\text{-}PREP\text{-}RECV\#\ outs)))\Longrightarrow Q$

and

not-in-err-exec32:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner)\ \Longrightarrow$
$\neg IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner$
$=$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV)\Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\ msg)))\ partner$
$\Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$

$thread\text{-}list \quad := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$
$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV))\|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}23\text{-}in\text{-}PREP\text{-}RECV\#\ outs))) \Longrightarrow Q$

and

$not\text{-}in\text{-}err\text{-}exec33:$
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\ caller\ partner\ \sigma\ msg \Longrightarrow$
$\neg IPC\text{-}params\text{-}c1\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c2\ ((the\ o\ thread\text{-}list\ \sigma)\ \ partner) \Longrightarrow$
$IPC\text{-}params\text{-}c6\ caller\ ((the\ o\ thread\text{-}list\ \sigma)\ partner) \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma'))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller$
$\Longrightarrow$
$th\text{-}flag\ \sigma = th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \Longrightarrow$
$(\sigma\|current\text{-}thread := caller,$
$thread\text{-}list \quad := update\text{-}th\text{-}ready\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes \quad := NO\text{-}ERRORS,$
$th\text{-}flag \quad := th\text{-}flag\ \sigma\|) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

shows Q
apply $(insert\ valid\text{-}exec)$
apply $(elim\ abort\text{-}prep\text{-}recv\text{-}mbindFSave\text{-}E')$
apply $(simp\ add{:}\ in\text{-}err\text{-}exec)$
apply $(simp\ only{:}\ \ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious3)$
apply $auto$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(rule\ not\text{-}in\text{-}err\text{-}exec1)$
apply $(simp\text{-}all\ add{:}\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious4)$
apply $auto$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(fold\ update\text{-}th\text{-}current.simps)$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add{:}\ not\text{-}in\text{-}err\text{-}exec2\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}fact0\text{-}def)$
apply $(simp\ add{:}\ exec\text{-}action_{id}\text{-}Mon\text{-}prep\text{-}recv\text{-}obvious5)$
apply $auto$
apply $(erule\ contrapos\text{-}np)$
apply $simp$
apply $(fold\ update\text{-}th\text{-}current.simps)$
apply $(subst\ (asm)\ threa\text{-}table\text{-}obvious')$
apply $(simp\ add{:}\ not\text{-}in\text{-}err\text{-}exec31)$

apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec32*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

P.3 Symbolic Execution rules for WAIT SEND

lemma *abort-wait-send-mbindFSave-E'*:
assumes *valid-exec*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)(*abort$_{lift}$ ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag σ*)) ⟹
(*σ* ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*get-caller-error caller σ # outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀*σ'*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*SEND caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ'*) ⟹
((*th-flag σ*)) *caller* = *None* ⟹
((*th-flag* (*error-tab-transfer caller σ σ'*))) *caller* =
((*th-flag σ*)) *caller* ⟹
th-flag (*error-tab-transfer caller σ σ'*) = *th-flag σ* ⟹
((*error-tab-transfer caller σ σ'*) ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*NO-ERRORS # outs*))) ⟹*Q*
and *not-in-err-state-Some2*:
⋀*σ' error-mem*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ'*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller* =
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *partner* =
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller* =
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *partner* ⟹
((*set-error-mem-waitr caller partner σ σ' error-mem msg*) ⊨
(*outs* ← (*mbind S*(*abort$_{lift}$ ioprog*)); *P* (*ERROR-MEM error-mem # outs*))) ⟹*Q*
and *not-in-err-state-Some3*:

$\bigwedge \sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* $\Longrightarrow$
((*set-error-ipc-waitr caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC WAIT* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-send-HOL-elim21'*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC WAIT* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))

and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$IPC\text{-}params\text{-}c5\ partner\ \sigma \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ \ = th\text{-}flag\ \ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$thread\text{-}list\ \ := update\text{-}th\text{-}waiting\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := NO\text{-}ERRORS,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#$
$outs))) \Longrightarrow Q$
and
not-in-err-exec21:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\ msg)))\ partner$
$\Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list\ \ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ \ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND,$
$th\text{-}flag\ \ := th\text{-}flag\ \sigma$
$(caller\ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND))|\!) \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}SEND\#\ outs))) \Longrightarrow Q$
and
not-in-err-exec22:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}send\text{-}comm\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$\neg IPC\text{-}params\text{-}c4\ caller\ partner \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$

$$\textit{error-IPC-3-in-WAIT-SEND msg}))\ \textit{caller} =$$
$$\textit{Some}\ (\textit{ERROR-IPC error-IPC-3-in-WAIT-SEND}) \Longrightarrow$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-3-in-WAIT-SEND msg}))\ \textit{partner} =$$
$$\textit{Some}\ (\textit{ERROR-IPC error-IPC-3-in-WAIT-SEND}) \Longrightarrow$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-3-in-WAIT-SEND msg}))\ \textit{caller} =$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-3-in-WAIT-SEND msg}))\ \textit{partner}$$
$$\Longrightarrow$$
$$(\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$$
$$\textit{thread-list} := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$$
$$\textit{error-codes} := \textit{ERROR-IPC error-IPC-3-in-WAIT-SEND},$$
$$\textit{th-flag} := \textit{th-flag}\ \sigma$$
$$(\textit{caller} \mapsto (\textit{ERROR-IPC error-IPC-3-in-WAIT-SEND}),$$
$$\textit{partner} \mapsto (\textit{ERROR-IPC error-IPC-3-in-WAIT-SEND}))|\!) \models$$
$$(\textit{outs} \leftarrow (\textit{mbind}\ S(\textit{abort}_{lift}\ \ \textit{exec-action}_{id}\textit{-Mon}));$$
$$P\ (\textit{ERROR-IPC error-IPC-3-in-WAIT-SEND\#}\ \textit{outs}))) \Longrightarrow Q$$

and

not-in-err-exec23:

$$\textit{caller} \notin \textit{dom}\ (\ (\textit{th-flag}\ \sigma)) \Longrightarrow$$
$$\textit{IPC-send-comm-check-st}_{id}\ \textit{caller partner}\ \sigma \Longrightarrow$$
$$\textit{IPC-params-c4 caller partner} \Longrightarrow$$
$$\neg \textit{IPC-params-c5 partner}\ \sigma \Longrightarrow$$
$$(\textit{thread-list}\ \sigma)\ \textit{caller} = \textit{None} \Longrightarrow$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-6-in-WAIT-SEND msg}))\ \textit{caller} =$$
$$\textit{Some}\ (\textit{ERROR-IPC error-IPC-6-in-WAIT-SEND}) \Longrightarrow$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-6-in-WAIT-SEND msg}))\ \textit{partner} =$$
$$\textit{Some}\ (\textit{ERROR-IPC error-IPC-6-in-WAIT-SEND}) \Longrightarrow$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-6-in-WAIT-SEND msg}))\ \textit{caller} =$$
$$(\ (\textit{th-flag}\ (\textit{set-error-ipc-maps caller partner}\ \sigma\ \sigma$$
$$\textit{error-IPC-6-in-WAIT-SEND msg}))\ \textit{partner}$$
$$\Longrightarrow$$
$$(\sigma(\!|\textit{current-thread} := \textit{caller}\ ,$$
$$\textit{thread-list} := \textit{update-th-current caller}\ (\textit{thread-list}\ \sigma),$$
$$\textit{error-codes} := \textit{ERROR-IPC error-IPC-6-in-WAIT-SEND},$$
$$\textit{th-flag} := \textit{th-flag}\ \sigma$$
$$(\textit{caller} \mapsto (\textit{ERROR-IPC error-IPC-6-in-WAIT-SEND}),$$
$$\textit{partner} \mapsto (\textit{ERROR-IPC error-IPC-6-in-WAIT-SEND}))|\!) \models$$
$$(\textit{outs} \leftarrow (\textit{mbind}\ S(\textit{abort}_{lift}\ \ \textit{exec-action}_{id}\textit{-Mon}));$$
$$P\ (\textit{ERROR-IPC error-IPC-6-in-WAIT-SEND\#}\ \textit{outs}))) \Longrightarrow Q$$

and

not-in-err-exec24:

$$\textit{caller} \notin \textit{dom}\ (\ (\textit{th-flag}\ \sigma)) \Longrightarrow$$
$$\textit{IPC-send-comm-check-st}_{id}\ \textit{caller partner}\ \sigma \Longrightarrow$$
$$\textit{IPC-params-c4 caller partner} \Longrightarrow$$

$\neg$*IPC-params-c5 partner* $\sigma \Longrightarrow$
$\exists$ *th*. (*thread-list* σ) *caller* = *Some th* $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *caller* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-SEND*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *partner* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-SEND*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-SEND msg))) *partner*
$\Longrightarrow$
(σ(|*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-SEND*,
th-flag := *th-flag* σ
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-SEND*),
partner $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-SEND*))|) $\models$
(*outs* $\leftarrow$ (*mbind* S(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-5-in-WAIT-SEND*# *outs*)))$\Longrightarrow$*Q*
shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-wait-send-mbindFSave-E'*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-send-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$-*Mon-def* *WAIT-SEND*$_{id}$-*def split*: *split-if-asm option.split-asm*)
apply (*auto*)
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-send-obvious4*)
apply *auto*
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*simp-all add*: *not-in-err-exec23*)
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*

apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-send-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

P.4 Symbolic Execution rules for WAIT RECV

lemma *abort-wait-recv-mbindFSave-E'*:
assumes *valid-exec*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC WAIT* (*RECV caller partner msg*))#*S*)(*abort$_{lift}$ ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag σ*)) ⟹
(*σ* ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*get-caller-error caller σ # outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀*σ'*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*RECV caller partner msg*)) *σ* = *Some*(*NO-ERRORS*, *σ'*) ⟹
((*th-flag σ*)) *caller* = *None* ⟹
((*th-flag* (*error-tab-transfer caller σ σ'*))) *caller* =
((*th-flag σ*)) *caller* ⟹
th-flag (*error-tab-transfer caller σ σ'*) = *th-flag σ* ⟹
((*error-tab-transfer caller σ σ'*) ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*NO-ERRORS # outs*))) ⟹*Q*
and *not-in-err-state-Some2*:
⋀*σ' error-mem*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC WAIT* (*RECV caller partner msg*)) *σ* = *Some*(*ERROR-MEM error-mem*, *σ'*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller* =
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *partner* =
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller* =
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *partner* ⟹
((*set-error-mem-waitr caller partner σ σ' error-mem msg*) ⊨
(*outs* ← (*mbind S*(*abort$_{lift}$ ioprog*)); *P* (*ERROR-MEM error-mem # outs*))) ⟹*Q*

and *not-in-err-state-Some3*:
$\bigwedge \sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*RECV caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* $\Longrightarrow$
((*set-error-ipc-waitr caller partner* σ σ' *error-IPC msg*) $\models$
(*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC WAIT* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC WAIT* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-wait-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-wait-recv-HOL-elim21'*:
assumes

$\mathit{valid\text{-}exec}$: $(\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ ((\mathit{IPC\ WAIT}\ (\mathit{RECV\ caller\ partner\ msg}))\#S)$
$(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));\ P\ \mathit{outs}))$

and $\mathit{in\text{-}err\text{-}exec}$:
$\mathit{caller} \in \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$(\sigma \models (\mathit{outs} \leftarrow (\mathit{mbind}\ \ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\mathit{get\text{-}caller\text{-}error\ caller}\ \sigma\ \#\ \mathit{outs}))) \Longrightarrow Q$

and
$\mathit{not\text{-}in\text{-}err\text{-}exec1}$:
$\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \Longrightarrow$
$\mathit{IPC\text{-}params\text{-}c4\ caller\ partner} \Longrightarrow$
$\mathit{IPC\text{-}params\text{-}c5\ partner}\ \sigma \Longrightarrow$
$(\ (\ \mathit{th\text{-}flag}\ \sigma))\ \mathit{caller} = \mathit{None} \Longrightarrow$
$(\ \ (\mathit{th\text{-}flag}\ (\mathit{error\text{-}tab\text{-}transfer\ caller}\ \sigma\ \sigma)))\ \mathit{caller}\ =$
$(\ \ (\mathit{th\text{-}flag}\ \sigma))\ \mathit{caller} \Longrightarrow$
$\mathit{th\text{-}flag}\ (\mathit{error\text{-}tab\text{-}transfer\ caller}\ \sigma\ \sigma)\ = \mathit{th\text{-}flag}\ \sigma\ \Longrightarrow$
$(\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller},$
$\mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}waiting\ caller}\ (\mathit{thread\text{-}list}\ \sigma),$
$\mathit{error\text{-}codes}\ \ := \mathit{NO\text{-}ERRORS},$
$\mathit{th\text{-}flag}\ \ := \mathit{th\text{-}flag}\ \sigma|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));\ P\ (\mathit{NO\text{-}ERRORS}\ \#\ \mathit{outs})))$
$\Longrightarrow Q$

and
$\mathit{not\text{-}in\text{-}err\text{-}exec21}$:
$\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\neg \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \Longrightarrow$
$(\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma$
$\mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV\ msg})))\ \mathit{caller} =$
$\mathit{Some}\ (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow$
$(\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma$
$\mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV\ msg})))\ \mathit{partner} =$
$\mathit{Some}\ (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow$
$(\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma$
$\mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV\ msg})))\ \mathit{caller} =$
$(\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner}\ \sigma\ \sigma$
$\mathit{error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV\ msg})))\ \mathit{partner} \Longrightarrow$
$(\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ ,$
$\mathit{thread\text{-}list}\ \ := \mathit{update\text{-}th\text{-}current\ caller}\ (\mathit{thread\text{-}list}\ \sigma),$
$\mathit{error\text{-}codes}\ \ := \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV},$
$\mathit{th\text{-}flag}\ \ := \mathit{th\text{-}flag}\ \sigma$
$(\mathit{caller}\ \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}),$
$\mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}))|\!) \models$
$(\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon}));$
$P\ (\ \mathit{ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}WAIT\text{-}RECV}\#\ \mathit{outs}))) \Longrightarrow Q$

and
$\mathit{not\text{-}in\text{-}err\text{-}exec22}$:
$\mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow$
$\mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller\ partner}\ \sigma \Longrightarrow$
$\neg \mathit{IPC\text{-}params\text{-}c4\ caller\ partner} \Longrightarrow$

$$
\begin{array}{l}
(\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{caller} = \\
\quad \mathit{Some}\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{partner} = \\
\quad \mathit{Some}\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{caller} = \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{partner} \\
\Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma \\
\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}), \\
\qquad\ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\qquad P\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}3\text{-}in\text{-}WAIT\text{-}RECV}\#\ \mathit{outs}))) \Longrightarrow Q
\end{array}
$$

and

$$
\begin{array}{l}
\mathit{not\text{-}in\text{-}err\text{-}exec23}: \\
\ \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\ \ \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\ \ \mathit{IPC\text{-}params\text{-}c4}\ \mathit{caller}\ \mathit{partner} \Longrightarrow \\
\ \neg \mathit{IPC\text{-}params\text{-}c5}\ \mathit{partner}\ \sigma \Longrightarrow \\
\ \ (\mathit{thread\text{-}list}\ \sigma)\ \mathit{caller} = \mathit{None} \Longrightarrow \\
\ \ (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{caller} = \\
\quad \mathit{Some}\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{partner} = \\
\quad \mathit{Some}\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}) \Longrightarrow \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{caller} = \\
\quad (\ \ (\mathit{th\text{-}flag}\ (\mathit{set\text{-}error\text{-}ipc\text{-}maps}\ \mathit{caller}\ \mathit{partner}\ \sigma\ \sigma \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}\ \mathit{msg})))\ \mathit{partner} \Longrightarrow \\
\quad (\sigma(\!|\mathit{current\text{-}thread} := \mathit{caller}\ , \\
\qquad \mathit{thread\text{-}list} \quad := \mathit{update\text{-}th\text{-}current}\ \mathit{caller}\ (\mathit{thread\text{-}list}\ \sigma), \\
\qquad \mathit{error\text{-}codes} \quad := \mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}, \\
\qquad \mathit{th\text{-}flag} \quad := \mathit{th\text{-}flag}\ \sigma \\
\qquad (\mathit{caller} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}), \\
\qquad\ \mathit{partner} \mapsto (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}))|\!) \models \\
\qquad (\mathit{outs} \leftarrow (\mathit{mbind}\ S(\mathit{abort}_{lift}\ \ \mathit{exec\text{-}action}_{id}\text{-}\mathit{Mon})); \\
\qquad\qquad P\ (\mathit{ERROR\text{-}IPC}\ \mathit{error\text{-}IPC\text{-}6\text{-}in\text{-}WAIT\text{-}RECV}\#\ \mathit{outs}))) \Longrightarrow Q
\end{array}
$$

and

$$
\begin{array}{l}
\mathit{not\text{-}in\text{-}err\text{-}exec24}: \\
\ \ \mathit{caller} \notin \mathit{dom}\ (\ (\mathit{th\text{-}flag}\ \sigma)) \Longrightarrow \\
\ \ \mathit{IPC\text{-}recv\text{-}comm\text{-}check\text{-}st}_{id}\ \mathit{caller}\ \mathit{partner}\ \sigma \Longrightarrow \\
\ \ \mathit{IPC\text{-}params\text{-}c4}\ \mathit{caller}\ \mathit{partner} \Longrightarrow
\end{array}
$$

$\neg$*IPC-params-c5 partner* $\sigma \Longrightarrow$
$\exists$ *th*. (*thread-list* σ) *caller* = *Some th* $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-RECV msg))) *caller* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-RECV*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-RECV msg))) *partner* =
Some (*ERROR-IPC error-IPC-5-in-WAIT-RECV*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-RECV msg))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ
error-IPC-5-in-WAIT-RECV msg))) *partner* $\Longrightarrow$
(σ($\|$*current-thread* := *caller* ,
thread-list := *update-th-current caller* (*thread-list* σ),
error-codes := *ERROR-IPC error-IPC-5-in-WAIT-RECV*,
th-flag := *th-flag* σ
(*caller* $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-RECV*),
partner $\mapsto$ (*ERROR-IPC error-IPC-5-in-WAIT-RECV*))$\|$) $\models$
(*outs* $\leftarrow$ (*mbind* S(*abort*$_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*ERROR-IPC error-IPC-5-in-WAIT-RECV*# *outs*)))$\Longrightarrow$*Q*

shows *Q*
apply (*insert valid-exec*)
apply (*elim abort-wait-recv-mbindFSave-E'*)
apply (*simp only*: *in-err-exec*)
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-recv-obvious3*)
apply (*simp add*: *not-in-err-exec1*)
apply (*simp add*: *exec-action*$_{id}$-*Mon-def WAIT-RECV*$_{id}$-*def split*: *split-if-asm option.split-asm*)
apply *auto*
apply (*simp only*: *exec-action*$_{id}$-*Mon-wait-recv-obvious4*)
apply (*auto*)
apply (*erule contrapos-np*)
apply (*simp*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *domIff*)
apply (*simp-all add*: *not-in-err-exec23*)
apply (*simp add*: *not-in-err-exec24*) +
apply (*erule contrapos-np*)
apply (*simp*)
apply (*fold update-th-current.simps*)
apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec22*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*erule contrapos-np*)
apply *simp*
apply (*fold update-th-current.simps*)

apply (*subst* (*asm*) *threa-table-obvious'*)
apply (*simp add*: *not-in-err-exec21*)
apply (*erule contrapos-np*)
apply *simp*
apply (*simp add*: *update-state-wait-recv-params5-def split*:*option.split-asm split-if-asm*)
apply (*simp add*: *exec-action$_{id}$-Mon-def*)
done

P.5 Symbolic Execution rules for BUF SEND

lemma *abort-buf-send-mbindFSave-E'*:
assumes *valid-exec*:
(*σ* ⊨ (*outs* ← (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)(*abort$_{lift}$ ioprog*));*P outs*))
and *in-err-state*:
caller ∈ *dom* ((*th-flag σ*)) ⟹
(*σ* ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*get-caller-error caller σ # outs*)))
⟹ *Q*
and *not-in-err-state-Some1*:
⋀*σ'*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
((*th-flag σ*)) *caller* = *None* ⟹
((*th-flag* (*error-tab-transfer caller σ σ'*))) *caller* =
((*th-flag σ*)) *caller* ⟹
th-flag (*error-tab-transfer caller σ σ'*) = *th-flag σ* ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*NO-ERRORS*,
σ') ⟹
((*error-tab-transfer caller σ σ'*) ⊨
(*outs* ← (*mbind S* (*abort$_{lift}$ ioprog*)); *P* (*NO-ERRORS # outs*))) ⟹*Q*
and *not-in-err-state-Some2*:
⋀*σ' error-mem*.
(*caller* ∉ *dom* ((*th-flag σ*))) ⟹
ioprog (*IPC BUF* (*SEND caller partner msg*)) *σ* = *Some*(*ERROR-MEM
error-mem*, *σ'*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller*
=
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*)))
partner =
Some (*ERROR-MEM error-mem*) ⟹
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*))) *caller*
=
((*th-flag* (*set-error-mem-maps caller partner σ σ' error-mem msg*)))
partner ⟹
((*set-error-mem-bufs caller partner σ σ' error-mem msg*)
⊨ (*outs* ← (*mbind S*(*abort$_{lift}$ ioprog*)); *P* (*ERROR-MEM error-mem
outs*))) ⟹*Q*
and *not-in-err-state-Some3*:

$\bigwedge \sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner*$\Longrightarrow$
((*set-error-ipc-bufs caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC BUF* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
using *valid-exec*
proof (*cases ioprog* (*IPC BUF* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *case-tac a*, *simp*,
simp split: *errors.split-asm*, *elim not-in-err-state-Some1*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-buf-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-buf-send-HOL-elim21'*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC BUF* (*SEND caller partner msg*))#*S*)

$$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$$

and *in-err-exec*:

$$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$$
$$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$$
$$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$$

and

not-in-err-exec1:

$$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$$
$$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$$
$$(\ (th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$$
$$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$$
$$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$$
$$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ = th\text{-}flag\ \sigma \Longrightarrow$$
$$(\sigma(\!|current\text{-}thread := caller,$$
$$resource\ \ := foldl\ (\lambda m\ (addr,val).\ (m\ (addr :=_{\$}\ val)))\ (resource\ \sigma)$$
$$(zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),$$
$$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$$
$$(update\text{-}th\text{-}ready\ partner$$
$$(thread\text{-}list\ \sigma)),$$
$$error\text{-}codes\ := NO\text{-}ERRORS,$$
$$th\text{-}flag\ \ := th\text{-}flag\ \sigma|\!))$$
$$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow$$
$$Rep\text{-}memory$$
$$(resource(\sigma(\!|current\text{-}thread := caller,$$
$$resource\ \ := foldl\ (\lambda m\ (addr,val).\ (m\ (addr :=_{\$}\ val)))\ (resource\ \sigma)$$
$$(zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),$$
$$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$$
$$(update\text{-}th\text{-}ready\ partner$$
$$(thread\text{-}list\ \sigma)),$$
$$error\text{-}codes\ := NO\text{-}ERRORS,$$
$$th\text{-}flag\ \ := th\text{-}flag\ \sigma|\!))) =$$
$$Rep\text{-}memory\ (foldl\ (\lambda m\ (addr,val).\ (m\ (addr :=_{\$}\ val)))\ (resource\ \sigma)$$
$$(zip\ (get\text{-}th\text{-}addrs\ partner\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma))) \Longrightarrow Q$$

and

not-in-err-exec12:

$$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$$
$$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow msg = [] \Longrightarrow$$
$$(\ (th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$$
$$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$$
$$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$$
$$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ = th\text{-}flag\ \sigma \Longrightarrow$$
$$(\sigma(\!|current\text{-}thread := caller,$$
$$resource\ \ := resource\ \sigma,$$
$$thread\text{-}list\ := update\text{-}th\text{-}ready\ caller$$
$$(update\text{-}th\text{-}ready\ partner$$

$(thread\text{-}list\ \sigma)),$
$error\text{-}codes\ := NO\text{-}ERRORS,$
$th\text{-}flag\ := th\text{-}flag\ \sigma\rbag$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$

and
not-in-err-exec2:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\ msg)))\ partner \Longrightarrow$
$(\sigma\lbag current\text{-}thread := caller\ ,$
$thread\text{-}list\ := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes\ := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND,$
$th\text{-}flag\ := th\text{-}flag\ \sigma$
$(caller\ \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND))\rbag \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}SEND\#\ outs))) \Longrightarrow Q$
shows Q
apply(*insert valid-exec*)
apply (*elim abort-buf-send-HOL-elim21*)
using *in-err-exec not-in-err-exec1 not-in-err-exec2 not-in-err-exec12*
apply *auto*
done

P.6 Symbolic Execution rules for BUF RECV

lemma *abort-buf-recv-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ BUF\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$

$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge \sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge \sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}ipc\text{-}bufr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\#\ outs))) \Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ BUF\ (RECV\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof $(cases\ caller \in dom\ (\ (th\text{-}flag\ \sigma)))$

```
  case True
  then show ?thesis
  using valid-exec
  by (subst (asm) abort-buf-recv-obvious10, elim in-err-state, simp)
next
  case False
  then show ?thesis
  using valid-exec
  proof (cases ioprog (IPC BUF (RECV caller partner msg)) σ)
    case (Some a)
    then show ?thesis
    using valid-exec False
    by (subst (asm) abort-buf-recv-obvious10, simp, case-tac a, simp,
        simp split: errors.split-asm, elim not-in-err-state-Some1,
        auto intro: not-in-err-state-Some2 not-in-err-state-Some3)
  next
    case None
    then show ?thesis
    using valid-exec False
    by (subst (asm) abort-buf-recv-obvious10, simp, elim not-in-err-state-None)
  qed
qed


lemma abort-buf-recv-HOL-elim21':
  assumes
    valid-exec: (σ ⊨ (outs ← (mbind ((IPC BUF (RECV caller partner msg))#S)
                          (abort_lift  exec-action_id-Mon)); P outs))
  and in-err-exec:
    caller ∈ dom ( (th-flag σ)) ⟹
         (σ ⊨ (outs ← (mbind  S(abort_lift  exec-action_id-Mon));
                              P (get-caller-error caller σ # outs))) ⟹ Q
  and
   not-in-err-exec1:
   caller ∉ dom ( (th-flag σ)) ⟹
    IPC-buf-check-st_id caller partner σ ⟹
    ( ( th-flag σ)) caller = None ⟹
    (  (th-flag (error-tab-transfer caller σ σ))) caller  =
    (  (th-flag σ)) caller ⟹
     th-flag (error-tab-transfer caller σ σ)  = th-flag σ ⟹
    (σ(|current-thread := caller,
         resource     :=foldl (λm (addr,val). (m (addr:=$ val))) (resource σ)
                              (zip (get-th-addrs caller σ) (get-msg-values msg σ)),
         thread-list  := update-th-ready caller
                         (update-th-ready partner
                         (thread-list σ)),
         error-codes  := NO-ERRORS,
         th-flag := th-flag σ|)
         ⊨ (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS #
```

$outs))) \Longrightarrow$
$Rep\text{-}memory$
$(resource(\sigma(\!|current\text{-}thread := caller,$
$resource := foldl\ (\lambda m\ (addr,val).\ (m\ (addr{:=}_{\$}\ val)))\ (resource\ \sigma)$
$(zip\ (get\text{-}th\text{-}addrs\ caller\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma)),$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!)) =$
$Rep\text{-}memory\ (foldl\ (\lambda m\ (addr,val).\ (m\ (addr{:=}_{\$}\ val)))\ (resource\ \sigma)$
$(zip\ (get\text{-}th\text{-}addrs\ caller\ \sigma)\ (get\text{-}msg\text{-}values\ msg\ \sigma))) \Longrightarrow Q$
and
$not\text{-}in\text{-}err\text{-}exec12$:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma \Longrightarrow$
$msg = [] \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)\ = th\text{-}flag\ \sigma \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller,$
$resource := resource\ \sigma,$
$thread\text{-}list := update\text{-}th\text{-}ready\ caller$
$(update\text{-}th\text{-}ready\ partner$
$(thread\text{-}list\ \sigma)),$
$error\text{-}codes := NO\text{-}ERRORS,$
$th\text{-}flag := th\text{-}flag\ \sigma|\!)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (NO\text{-}ERRORS\ \#$
$outs))) \Longrightarrow Q$

and
$not\text{-}in\text{-}err\text{-}exec2$:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$\neg\ IPC\text{-}buf\text{-}check\text{-}st_{id}\ caller\ partner\ \sigma\ \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ partner =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma$
$error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\ msg)))\ partner \Longrightarrow$
$(\sigma(\!|current\text{-}thread := caller\ ,$
$thread\text{-}list := update\text{-}th\text{-}current\ caller\ (thread\text{-}list\ \sigma),$
$error\text{-}codes := ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV,$
$th\text{-}flag := th\text{-}flag\ \sigma$

$(caller \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV),$
$partner \mapsto (ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV))\rrbracket \models$
$(outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (\ ERROR\text{-}IPC\ error\text{-}IPC\text{-}1\text{-}in\text{-}BUF\text{-}RECV\#\ outs)))\Longrightarrow Q$
shows Q
apply(*insert valid-exec*)
apply (*elim abort-buf-recv-HOL-elim21*)
using *in-err-exec not-in-err-exec1 not-in-err-exec2 not-in-err-exec12*
apply *auto*
done

P.7 Symbolic Execution rules for MAP SEND

lemma *abort-map-send-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma\ \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (SEND\ caller\ partner\ msg))\ \sigma =\ \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(\ \ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$

$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-MEM error-mem* # *outs*))) $\Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'$ *error-IPC*.
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *Some*(*ERROR-IPC error-IPC*, σ') $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
Some (*ERROR-IPC error-IPC*) $\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* =
Some (*ERROR-IPC error-IPC*)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner* $\Longrightarrow$
((*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow Q$
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*SEND caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-send-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC MAP* (*SEND caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-send-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-send-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *mem-share-list-E*:

assumes *1*: $resource\ \sigma' =$
$(foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ (resource\ \sigma)\ (n\#ns))$
and *2*: $(fst\ n)\ shares_{(resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))}\ (snd\ n) \Longrightarrow$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (fst\ n)) =$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (snd\ n)) \Longrightarrow$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (fst\ n)) = ((resource\ \sigma)\ \$\ (fst\ n))$
$\Longrightarrow$
$resource\ \sigma' =$
$(foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ ((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n)))$
$ns) \Longrightarrow Q$
shows *Q*
using *1*
using *transfer-share* [*of fst n*, *of* (*resource* σ), *of snd n*]
transfer-share-lookup2[*of* (*resource* σ), *of fst n*, *of snd n*]
transfer-share-lookup1 [*of* (*resource* σ), *of fst n*, *of snd n*]
apply (*elim 2*)
apply (*simp-all add*: *Product-Type.split-beta*)
done

lemma *mem-share-list-I*:
$(fst\ n)\ shares_{(resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))}\ (snd\ n) \Longrightarrow$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (fst\ n)) =$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (snd\ n)) \Longrightarrow$
$((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n))\ \$\ (fst\ n)) = ((resource\ \sigma)\ \$\ (fst\ n))$
$\Longrightarrow$
$resource\ \sigma' =$
$(foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ ((resource\ \sigma)((fst\ n) \bowtie\ (snd\ n)))$
$ns) \Longrightarrow$
$resource\ \sigma' =$
$(foldl\ (\lambda m\ (src,dst).\ (m\ (src \bowtie\ dst)))\ (resource\ \sigma)\ (n\#ns))$
using *transfer-share* [*of fst n*, *of* (*resource* σ), *of snd n*]
transfer-share-lookup2[*of* (*resource* σ), *of fst n*, *of snd n*]
transfer-share-lookup1 [*of* (*resource* σ), *of fst n*, *of snd n*]
apply (*simp-all add*: *Product-Type.split-beta*)
done

lemma *abort-map-send-HOL-elim2'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (SEND\ caller\ partner\ msg))\#S)$
$(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-exec*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ \ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));$
$P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow Q$
and
not-in-err-exec1:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma)))\ caller =$

```
   (  (th-flag σ)) caller ⟹
   th-flag (error-tab-transfer caller σ σ)  = th-flag σ  ⟹
   (σ(|current-thread := caller,
      resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                              (zip msg (get-th-addrs partner σ)),
      thread-list    := update-th-ready caller
                        (update-th-ready partner
                        (thread-list σ)),
      error-codes    := NO-ERRORS,
      th-flag        := th-flag σ|) ⊨
   (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS # outs)))
⟹

     (resource(σ(|current-thread := caller,
                 resource       := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                                         (zip msg (get-th-addrs partner σ)),
                 thread-list    := update-th-ready caller
                                   (update-th-ready partner
                                   (thread-list σ)),
                 error-codes    := NO-ERRORS,
                 th-flag        := th-flag σ|))) =
       (foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
                    (zip msg (get-th-addrs partner σ)))⟹ Q

  and
   not-in-err-exec12:
   caller ∉ dom ( (th-flag σ)) ⟹
   ( ( th-flag σ)) caller = None ⟹
   (  (th-flag (error-tab-transfer caller σ σ))) caller =
   (  (th-flag σ)) caller ⟹
   th-flag (error-tab-transfer caller σ σ)  = th-flag σ  ⟹
   msg = [] ⟹
   (σ(|current-thread := caller,
      resource       := resource σ,
      thread-list    := update-th-ready caller
                        (update-th-ready partner
                        (thread-list σ)),
      error-codes    := NO-ERRORS,
      th-flag        := th-flag σ|) ⊨
   (outs ← (mbind S(abort_lift  exec-action_id-Mon)); P (NO-ERRORS # outs)))
⟹ Q

  shows Q
  apply(insert valid-exec)
  apply (elim abort-map-send-HOL-elim2)
  using in-err-exec not-in-err-exec1 not-in-err-exec12
  apply auto
  done
```

P.8 Symbolic Execution rules for MAP RECV

lemma *abort-map-recv-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ MAP\ (RECV\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some1*:
$\bigwedge\sigma'.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = Some(NO\text{-}ERRORS, \sigma') \Longrightarrow$
$(\ (\ th\text{-}flag\ \sigma))\ caller = None \Longrightarrow$
$(\ \ (th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')))\ caller\ =$
$(\ \ (th\text{-}flag\ \sigma))\ caller \Longrightarrow$
$th\text{-}flag\ (error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma')\ = th\text{-}flag\ \sigma \Longrightarrow$
$((error\text{-}tab\text{-}transfer\ caller\ \sigma\ \sigma') \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some2*:
$\bigwedge\sigma'\ error\text{-}mem.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}MEM\ error\text{-}mem, \sigma') \Longrightarrow$
$(\ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner =$
$Some\ (ERROR\text{-}MEM\ error\text{-}mem) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ caller =$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}mem\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)))\ partner \Longrightarrow$
$((set\text{-}error\text{-}mem\text{-}mapr\ caller\ partner\ \sigma\ \sigma'\ error\text{-}mem\ msg)$
$\models (outs \leftarrow (mbind\ S(abort_{lift}\ \ ioprog));\ P\ (ERROR\text{-}MEM\ error\text{-}mem\ \#\ outs))) \Longrightarrow Q$
and *not-in-err-state-Some3*:
$\bigwedge\sigma'\ error\text{-}IPC.$
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ MAP\ (RECV\ caller\ partner\ msg))\ \sigma = \ Some(ERROR\text{-}IPC\ error\text{-}IPC, \sigma') \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ caller =$
$Some\ (ERROR\text{-}IPC\ error\text{-}IPC) \Longrightarrow$
$(\ \ (th\text{-}flag\ (set\text{-}error\text{-}ipc\text{-}maps\ caller\ partner\ \sigma\ \sigma'\ error\text{-}IPC\ msg)))\ partner =$

Some (ERROR-IPC error-IPC)$\Longrightarrow$
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *caller* =
((*th-flag* (*set-error-ipc-maps caller partner* σ σ' *error-IPC msg*))) *partner*
$\Longrightarrow$
((*set-error-ipc-mapr caller partner* σ σ' *error-IPC msg*)
$\models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *ioprog*)); *P* (*ERROR-IPC error-IPC*# *outs*))) $\Longrightarrow$ *Q*
and *not-in-err-state-None*:
(*caller* $\notin$ *dom* ((*th-flag* σ))) $\Longrightarrow$
ioprog (*IPC MAP* (*RECV caller partner msg*)) σ = *None* $\Longrightarrow$
($\sigma \models$ (*P* [])) $\Longrightarrow$ *Q*
shows *Q*
proof (*cases caller* $\in$ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *elim in-err-state*, *simp*)
next
case *False*
then show *?thesis*
proof (*cases ioprog* (*IPC MAP* (*RECV caller partner msg*)) σ)
case (*Some a*)
then show *?thesis*
using *valid-exec False Some*
by (*subst* (*asm*) *abort-map-recv-obvious10*,
case-tac a,*simp split*: *errors.split-asm*, *simp*, *elim not-in-err-state-Some1*, *simp*,
auto intro: *not-in-err-state-Some2 not-in-err-state-Some3*)
next
case *None*
then show *?thesis*
using *valid-exec False*
by (*subst* (*asm*) *abort-map-recv-obvious10*, *simp*, *elim not-in-err-state-None*)
qed
qed

lemma *abort-map-recv-HOL-elim2'*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC MAP* (*RECV caller partner msg*))#*S*)
($abort_{lift}$ *exec-action*$_{id}$-*Mon*)); *P outs*))
and *in-err-exec*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* $S(abort_{lift}$ *exec-action*$_{id}$-*Mon*));
P (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*
and
not-in-err-exec1:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
((*th-flag* σ)) *caller* = *None* $\Longrightarrow$

((th-flag (error-tab-transfer caller σ σ))) caller =
((th-flag σ)) caller ⟹
th-flag (error-tab-transfer caller σ σ) = th-flag σ ⟹
(σ(|current-thread := caller,
resource := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
(zip msg (get-th-addrs caller σ)),
thread-list := update-th-ready caller
(update-th-ready partner
(thread-list σ)),
error-codes := NO-ERRORS,
th-flag := th-flag σ|)
⊨ (outs ← (mbind S(abort$_{lift}$ exec-action$_{id}$-Mon)); P (NO-ERRORS # outs))) ⟹
Rep-memory
(resource(σ(|current-thread := caller,
resource := foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
(zip msg (get-th-addrs caller σ)),
thread-list := update-th-ready caller
(update-th-ready partner
(thread-list σ)),
error-codes := NO-ERRORS,
th-flag := th-flag σ|))) =
Rep-memory (foldl (λm (src,dst). (m (src⋈ dst))) (resource σ)
(zip msg (get-th-addrs caller σ))) ⟹ Q

and
not-in-err-exec12:
caller ∉ dom ((th-flag σ)) ⟹ msg = [] ⟹
((th-flag σ)) caller = None ⟹
((th-flag (error-tab-transfer caller σ σ))) caller =
((th-flag σ)) caller ⟹
th-flag (error-tab-transfer caller σ σ) = th-flag σ ⟹
(σ(|current-thread := caller,
resource := resource σ,
thread-list := update-th-ready caller
(update-th-ready partner
(thread-list σ)),
error-codes := NO-ERRORS,
th-flag := th-flag σ|)
⊨ (outs ← (mbind S(abort$_{lift}$ exec-action$_{id}$-Mon)); P (NO-ERRORS # outs))) ⟹ Q
shows Q
apply(insert valid-exec)
apply (elim abort-map-recv-HOL-elim2)
using in-err-exec not-in-err-exec1 not-in-err-exec12
apply auto
done

P.9 Symbolic Execution rules for DONE SEND

lemma *abort-done-send-mbindFSave-E'*:
assumes *valid-exec*:
$(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (SEND\ caller\ partner\ msg))\#S)(abort_{lift}\ ioprog));P\ outs))$
and *in-err-state1*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow\ caller \neq partner \Longrightarrow$
$((\ (th\text{-}flag\ \sigma))\ partner =$
$((\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner) \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *in-err-state2*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow\ caller = partner \Longrightarrow$
$((\ (th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma)))\ )\ partner = None \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-Some*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma \neq None \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S\ (abort_{lift}\ \ ioprog));\ P\ (NO\text{-}ERRORS\ \#\ outs)))$
$\Longrightarrow Q$
and *not-in-err-state-None*:
$(caller \notin dom\ (\ (th\text{-}flag\ \sigma))) \Longrightarrow$
$ioprog\ (IPC\ DONE\ (SEND\ caller\ partner\ msg))\ \sigma = None \Longrightarrow$
$(\sigma \models (P\ [])) \Longrightarrow Q$
shows Q
proof (*cases* $caller \in dom\ (\ (th\text{-}flag\ \sigma))$)
case *True*
then show *?thesis*
using *valid-exec*
apply (*subst* (*asm*) *abort-done-send-obvious12*, *simp*)
apply (*erule disjE*)
apply (*erule conjE*)+
apply (*simp add*: *in-err-state1*)
apply (*erule conjE*)+
apply (*simp add*: *in-err-state2*)
done
next
case *False*
assume *hyp1*: $caller \notin dom\ (\ (th\text{-}flag\ \sigma))$
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*SEND caller partner msg*)) $\sigma \neq None$)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-send-obvious11*, *simp only*: *False comp-apply*)
next

case *False*
then show *?thesis*
using *valid-exec False hyp1*
apply (*subst* (*asm*) *abort-done-send-obvious11*)
apply (*simp only*: *if-False comp-apply split*: *bool.split-asm*)
apply (*elim not-in-err-state-None*)
apply (*erule contrapos-np*)
apply (*simp-all*)
done
qed
qed

lemma *abort-done-send-HOL-elim1'*:
assumes
valid-exec: ($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*SEND caller partner msg*))#*S*)
($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P outs*))

and *in-err-state1*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$ *caller* $\neq$ *partner* $\Longrightarrow$
(((*th-flag* (*remove-caller-error caller* σ)))) *partner* =
((*th-flag* σ)) *partner* $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*)))
$\Longrightarrow$ *Q*

and *in-err-state2*:
caller $\in$ *dom* ((*th-flag* σ)) $\Longrightarrow$ *caller* = *partner* $\Longrightarrow$
(((*th-flag* (*remove-caller-error caller* σ)))) *partner* = *None* $\Longrightarrow$
((*remove-caller-error caller* σ) $\models$
(*outs* $\leftarrow$ (*mbind S* ($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*)); *P* (*get-caller-error caller* σ # *outs*))) $\Longrightarrow$ *Q*

and *not-in-err-exec1*:
caller $\notin$ *dom* ((*th-flag* σ)) $\Longrightarrow$
($\sigma \models$ (*outs* $\leftarrow$ (*mbind S*($abort_{lift}$ $exec\text{-}action_{id}$-*Mon*));*P* (*NO-ERRORS* # *outs*))) $\Longrightarrow$ *Q*

shows *Q*
using *valid-exec*
by (*rule abort-done-send-mbindFSave-E'*,
simp-all add: $exec\text{-}action_{id}$-*Mon-def in-err-state1 in-err-state2 not-in-err-exec1*)

P.10 Symbolic Execution rules for DONE SEND

lemma *abort-done-recv-mbindFSave-E'*:
assumes *valid-exec*:
($\sigma \models$ (*outs* $\leftarrow$ (*mbind* ((*IPC DONE* (*RECV caller partner msg*))#*S*)($abort_{lift}$ *ioprog*));*P outs*))

and *in-err-state1*:
caller ∈ *dom* ((*th-flag* σ)) ⟹ *caller* ≠ *partner* ⟹
(((*th-flag* σ)) *partner* =
(((*th-flag* (*remove-caller-error caller* σ)))) *partner*) ⟹
((*remove-caller-error caller* σ) ⊨
(*outs* ← (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
⟹ *Q*

and *in-err-state2*:
caller ∈ *dom* ((*th-flag* σ)) ⟹ *caller* = *partner* ⟹
(((*th-flag* (*remove-caller-error caller* σ)))) *partner* = *None* ⟹
((*remove-caller-error caller* σ) ⊨
(*outs* ← (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*get-caller-error caller* σ # *outs*)))
⟹ *Q*

and *not-in-err-state-Some*:
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None* ⟹
(σ ⊨ (*outs* ← (*mbind S* ($abort_{lift}$ *ioprog*)); *P* (*NO-ERRORS* # *outs*)))
⟹ *Q*

and *not-in-err-state-None*:
(*caller* ∉ *dom* ((*th-flag* σ))) ⟹
ioprog (*IPC DONE* (*RECV caller partner msg*)) σ = *None* ⟹
(σ ⊨ (*P* [])) ⟹ *Q*

shows *Q*
proof (*cases caller* ∈ *dom* ((*th-flag* σ)))
case *True*
then show *?thesis*
using *valid-exec*
apply (*subst* (*asm*) *abort-done-recv-obvious12*, *simp*)
apply (*erule disjE*)
apply (*erule conjE*)+
apply (*simp add*: *in-err-state1*)
apply (*erule conjE*)+
apply (*simp add*: *in-err-state2*)
done
next
case *False*
assume *hyp1*: *caller* ∉ *dom* ((*th-flag* σ))
then show *?thesis*
proof (*cases ioprog* (*IPC DONE* (*RECV caller partner msg*)) σ ≠ *None*)
case *True*
then show *?thesis*
using *assms*
by (*subst* (*asm*) *abort-done-recv-obvious11*, *simp only*: *False*)
next

case *False*
then show *?thesis*
using *valid-exec False hyp1*
apply (*subst* (*asm*) *abort-done-recv-obvious11*)
apply (*simp only*: *if-False split*: *bool.split-asm*)
apply (*elim not-in-err-state-None*)
apply (*erule contrapos-np*)
apply (*simp-all*)
done
qed
qed

lemma *abort-done-recv-HOL-elim1'*:
assumes
valid-exec: $(\sigma \models (outs \leftarrow (mbind\ ((IPC\ DONE\ (RECV\ caller\ partner\ msg))\#S)\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ outs))$
and *in-err-state1*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow caller \neq partner \Longrightarrow$
$(th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ partner =$
$(th\text{-}flag\ \sigma)\ partner \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs)))$
$\Longrightarrow Q$
and *in-err-state2*:
$caller \in dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow\ caller = partner \Longrightarrow$
$(th\text{-}flag\ (remove\text{-}caller\text{-}error\ caller\ \sigma))\ partner = None \Longrightarrow$
$((remove\text{-}caller\text{-}error\ caller\ \sigma) \models$
$(outs \leftarrow (mbind\ S\ (abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));\ P\ (get\text{-}caller\text{-}error\ caller\ \sigma\ \#\ outs))) \Longrightarrow$
Q
and *not-in-err-exec1*:
$caller \notin dom\ (\ (th\text{-}flag\ \sigma)) \Longrightarrow$
$(\sigma \models (outs \leftarrow (mbind\ S(abort_{lift}\ \ exec\text{-}action_{id}\text{-}Mon));P\ (NO\text{-}ERRORS\ \#\ outs))) \Longrightarrow Q$
shows Q
using *valid-exec*
by (*rule abort-done-recv-mbindFSave-E'*,
simp-all add: $exec\text{-}action_{id}$*-Mon-def in-err-state1 in-err-state2 not-in-err-exec1*)

end

theory *IPC-system-calls*

imports *IPC-symbolic-exec-intros IPC-symbolic-exec-elims*

begin

Q HOL representation of PikeOS IPC system calls

We define a system call by a set of operations. PikeOS IPC API contain 7 system calls, each system call can do a set of operations. In this section we will just present the most general one called *p4_ipc*:

type-synonym $behaviour_{ipc} = trace_{ipc}\ set$
type-synonym $behaviour_{ipc}{}' = trace_{ipc}\ list$

Q.1 System calls with thread ID as argument

type-synonym $behaviour_{id} = trace_{ipc}\ list$

definition $P4\text{-}IPC\text{-}BUF_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC\text{-}BUF_{id}\ caller\ partner\ msg \equiv$
$[caller \rhd_{id} msg \rhd_{id} partner,\ caller \lhd_{id} msg \lhd_{id} partner,$
$caller \trianglerighteq_{id} msg \trianglerighteq_{id} partner,\ caller \trianglelefteq_{id} msg \trianglelefteq_{id} partner]$

definition $P4\text{-}IPC\text{-}BUF\text{-}SEND_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC\text{-}BUF\text{-}SEND_{id}\ caller\ partner\ msg \equiv [caller \rhd_{id} msg \rhd_{id} partner,\ caller \trianglerighteq_{id} msg \trianglerighteq_{id} partner]$

definition $P4\text{-}IPC\text{-}BUF\text{-}RECV_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC\text{-}BUF\text{-}RECV_{id}\ caller\ partner\ msg \equiv [caller \lhd_{id} msg \lhd_{id} partner,\ caller \trianglelefteq_{id} msg \trianglelefteq_{id} partner]$

definition $P4\text{-}IPC\text{-}SEND_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC\text{-}SEND_{id}\ caller\ partner\ msg \equiv [caller \rhd_{id} msg \rhd_{id} partner,\ caller \trianglerighteq_{id} msg \trianglerighteq_{id} partner]$

definition $P4\text{-}IPC\text{-}RECV_{id}$
$::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC\text{-}RECV_{id}\ caller\ partner\ msg \equiv [caller \lhd_{id} msg \lhd_{id} partner,\ caller \trianglelefteq_{id}$

$msg \trianglelefteq_{id} partner]$

definition $P4\text{-}IPC_{id}$
$\quad ::thread_{id} \Rightarrow thread_{id} \Rightarrow int\ list \Rightarrow behaviour_{id}$
where
$P4\text{-}IPC_{id}\ caller\ partner\ msg \equiv$
$\quad [caller \triangleright_{id} msg \triangleright_{id} partner,\ caller \triangleleft_{id} msg \triangleleft_{id} partner,$
$\quad\ caller \trianglerighteq_{id} msg \trianglerighteq_{id} partner,\ caller \trianglelefteq_{id} msg \trianglelefteq_{id} partner]$

Q.2 System calls based on datatype

datatype $('thread\text{-}id,\ 'msg)\ P4\text{-}IPC\text{-}call =$
$\quad P4\text{-}IPC\text{-}call \quad 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}SEND\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}RECV\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}BUF\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}MAP\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$
$|\ P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ 'thread\text{-}id\ 'thread\text{-}id\ 'msg$

value $int(card(interleave\ ([IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$\quad IPC\ WAIT\ (SEND\ caller\ partner\ msg),$
$\quad IPC\ BUF\ \ (SEND\ caller\ partner\ msg),$
$\quad IPC\ MAP\ \ (SEND\ caller\ partner\ msg),$
$\quad IPC\ DONE\ (SEND\ caller\ partner\ msg)])$
$\quad ([IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$\quad IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$\quad IPC\ BUF\ \ (RECV\ caller\ partner\ msg),$
$\quad IPC\ MAP\ \ (RECV\ caller\ partner\ msg),$
$\quad IPC\ DONE\ (RECV\ caller\ partner\ msg)])))$

fun $IPC\text{-}call\text{-}sem::('thread\text{-}id,\ 'msg)\ P4\text{-}IPC\text{-}call \Rightarrow$
$\quad ((p4\text{-}stage_{ipc}, ('thread\text{-}id\ ,\ 'msg\ )\ p4\text{-}direct_{ipc})action_{ipc}\ list)$
where
$IPC\text{-}call\text{-}sem\ (P4\text{-}IPC\text{-}call\ caller\ partner\ msg) =$
$\quad ([IPC\ PREP\ (SEND\ caller\ partner\ msg),$
$\quad IPC\ WAIT\ (SEND\ caller\ partner\ msg),$
$\quad IPC\ BUF\ \ (SEND\ caller\ partner\ msg),$
$\quad IPC\ MAP\ \ (SEND\ caller\ partner\ msg),$
$\quad IPC\ DONE\ (SEND\ caller\ partner\ msg),$
$\quad IPC\ PREP\ (RECV\ caller\ partner\ msg),$
$\quad IPC\ WAIT\ (RECV\ caller\ partner\ msg),$
$\quad IPC\ BUF\ \ (RECV\ caller\ partner\ msg),$
$\quad IPC\ MAP\ \ (RECV\ caller\ partner\ msg),$
$\quad IPC\ DONE\ (RECV\ caller\ partner\ msg)])|$

```
IPC-call-sem (P4-IPC-SEND-call caller partner msg)     =
          ([IPC PREP (SEND caller partner msg),
            IPC WAIT (SEND caller partner msg),
            IPC BUF  (SEND caller partner msg),
            IPC MAP  (SEND caller partner msg),
            IPC DONE (SEND caller partner msg)])
                  |
IPC-call-sem (P4-IPC-RECV-call caller partner msg)     =
                ([IPC PREP (RECV caller partner msg),
                  IPC WAIT (RECV caller partner msg),
                  IPC BUF  (RECV caller partner msg),
                  IPC MAP  (RECV caller partner msg),
                  IPC DONE (RECV caller partner msg)])|
IPC-call-sem (P4-IPC-BUF-call caller partner msg)      =
           ([IPC PREP (SEND caller partner msg),
             IPC WAIT (SEND caller partner msg),
             IPC BUF  (SEND caller partner msg),
             IPC DONE (SEND caller partner msg),
             IPC PREP (RECV caller partner msg),
             IPC WAIT (RECV caller partner msg),
             IPC BUF  (RECV caller partner msg),
             IPC DONE (RECV caller partner msg)])|
IPC-call-sem (P4-IPC-BUF-SEND-call caller partner msg) =
         ([IPC PREP (SEND caller partner msg),
           IPC WAIT (SEND caller partner msg),
           IPC BUF  (SEND caller partner msg),
           IPC DONE (SEND caller partner msg)])
                  |
IPC-call-sem (P4-IPC-BUF-RECV-call caller partner msg) =
           ([IPC PREP (RECV caller partner msg),
             IPC WAIT (RECV caller partner msg),
             IPC BUF  (RECV caller partner msg),
             IPC DONE (RECV caller partner msg)])|
IPC-call-sem (P4-IPC-MAP-call caller partner msg)      =
          ([IPC PREP (SEND caller partner msg),
            IPC WAIT (SEND caller partner msg),
            IPC MAP  (SEND caller partner msg),
            IPC DONE (SEND caller partner msg),
            IPC PREP (RECV caller partner msg),
            IPC WAIT (RECV caller partner msg),
            IPC MAP  (RECV caller partner msg),
            IPC DONE (RECV caller partner msg)])|
IPC-call-sem (P4-IPC-MAP-SEND-call caller partner msg) =
          ([IPC PREP (SEND caller partner msg),
            IPC WAIT (SEND caller partner msg),
            IPC MAP  (SEND caller partner msg),
            IPC DONE (SEND caller partner msg)])|
IPC-call-sem (P4-IPC-MAP-RECV-call caller partner msg) =
           ([IPC PREP (RECV caller partner msg),
```

IPC WAIT (*RECV caller partner msg*),
IPC MAP (*RECV caller partner msg*),
IPC DONE (*RECV caller partner msg*)])

Q.3 Predicates on system calls

definition *is-ipc-system-call*$_{id}$
where *is-ipc-system-call*$_{id}$ *sc* = (∃ *caller partner msg*. *sc* = *P4-IPC*$_{id}$ *caller partner msg*)

lemmas *system-calls-normalizer* =
is-ipc-system-call$_{id}$*-def P4-IPC*$_{id}$*-def*

end

theory *IPC-coverage*

imports *IPC-system-calls*

begin

fun *sync-communication*
:: *'a list* ⇒*'a list* ⇒ *'a list* ⇒ *'a list set* ((- /⌊-⌋/ -) [*201*, *0*, *201*] *200*)
where
[] ⌊[]⌋ [] = {[]}|
A ⌊[]⌋ *B* = *interleave A B*|
[] ⌊*N*⌋ [] = {[]}|
A ⌊[*n1*, *n2*]⌋ [] = (*if n1* ∈ *set A* ∨ *n2* ∈ *set A then* {} *else* {*A*})|
[] ⌊[*n1*,*n2*]⌋ (*B*) = (*if n1* ∈ *set B* ∨ *n2* ∈ *set B then* {} *else* {*B*})|
(*a*#*A*) ⌊[*n1*,*n2*]⌋ (*b*#*B*) = (*if* (*a* = *n1* ∧ *b* = *n2*)
then image (λ *x*. *n1* #*n2*# *x*) (*A* ⌊[*n1*,*n2*]⌋ *B*)
else
if a ≠ *n1* ∧ *b* = *n2*
then image (λ *x*. *a* # *x*) (*A* ⌊[*n1*,*n2*]⌋ (*b*#*B*))
else
if a = *n1* ∧ *b* ≠ *n2*
then image (λ *x*. *b* # *x*) ((*a*#*A*) ⌊[*n1*,*n2*]⌋ *B*)
else (*image*(λ *x*. *a* # *x*) (*A* ⌊[*n1*,*n2*]⌋ (*b*#*B*)) ∪
(*image* (λ *x*. *b* # *x*) ((*a*#*A*) ⌊[*n1*,*n2*]⌋ *B*)))))|
A ⌊*N*⌋ *B* = *A* ⌊[]⌋ *B*

datatype (*'th-id*, *'sclist*)*criterion* =
interleave-all (*'th-id* ×*'sclist*) *list*
|*TPAIR 'th-id 'th-id 'th-id* ⇀*'sclist*
|*COMM 'th-id 'th-id 'th-id* ⇀*'sclist*

Q.4 Derivation of communication from system calls

— Definition that let us to derive PikeOS ipc communication from the different system calls

definition
[*simp*]:
sc-cases-IPC-call th msg th′ sc′ =
(*case sc′ of P4-IPC-call th1′ th2′′ msg′* ⇒
(*if* (*th2′′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*) (∗*check if th′ is caller of sc′ and th is his partner and msg msg′ are equal*∗)
then {}
else ((*th* ▷ *msg* ▷ *th′*)
⌊[*IPC WAIT* (*SEND th th′ msg*) , *IPC WAIT* (*RECV th′ th msg*)]⌋
(*th′*◁ *msg* ◁ *th*) ∪
(*th′* ◁ *msg* ◁ *th*)
⌊[*IPC WAIT* (*RECV th′ th msg*) , *IPC WAIT* (*SEND th th′ msg*)]⌋
(*th* ▷ *msg* ▷ *th′*) ∪
(*th′* ▷ *msg* ▷ *th*)
⌊[*IPC WAIT* (*SEND th′ th msg*) , *IPC WAIT* (*RECV th th′ msg*)]⌋
(*th*◁ *msg* ◁ *th′*) ∪
(*th* ◁ *msg* ◁ *th′*)
⌊[*IPC WAIT* (*RECV th th′ msg*) , *IPC WAIT* (*SEND th′ th msg*)]⌋
(*th′* ▷ *msg* ▷ *th*)))
|*P4-IPC-SEND-call th1′ th2′′ msg′* ⇒
(*if* (*th2′′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*)
then {}
else ((*th′* ▷ *msg* ▷ *th*)
⌊[*IPC WAIT* (*SEND th′ th msg*) , *IPC WAIT* (*RECV th th′ msg*)]⌋
(*th*◁ *msg* ◁ *th′*) ∪
(*th* ◁ *msg* ◁ *th′*)
⌊[*IPC WAIT* (*RECV th th′ msg*) , *IPC WAIT* (*SEND th′ th msg*)]⌋
(*th′* ▷ *msg* ▷ *th*)))
|*P4-IPC-RECV-call th1′ th2′′ msg′* ⇒
(*if* (*th2′′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*)
then {}
else (*th* ▷ *msg* ▷ *th′*)
⌊[*IPC WAIT* (*SEND th th′ msg*) , *IPC WAIT* (*RECV th′ th msg*)]⌋
(*th′*◁ *msg* ◁ *th*) ∪
(*th′* ◁ *msg* ◁ *th*)
⌊[*IPC WAIT* (*RECV th′ th msg*) , *IPC WAIT* (*SEND th th′ msg*)]⌋
(*th* ▷ *msg* ▷ *th′*))
|*P4-IPC-BUF-call th1′ th2′′ msg′* ⇒
(*if* (*th2′′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*)
then {}
else (((*th* ▷ *msg* ▷ *th′*)
⌊[*IPC WAIT* (*SEND th th msg*) , *IPC WAIT* (*RECV th′ th msg*)]⌋
([*IPC PREP* (*RECV th′ th msg*), *IPC WAIT* (*RECV th′ th msg*),
IPC BUF (*RECV th′ th msg*), *IPC DONE* (*SEND th′ th msg*),

$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$

$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2''\ msg' \Rightarrow$
$(if\ \ (th2'' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$

$|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1'\ th2''\ msg' \Rightarrow$
$(if\ \ (th2'' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$

$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2''\ msg' \Rightarrow$
$(if\ \ (th2'' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$

$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2''\ msg' \Rightarrow$
$(if\ \ (th2'' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th1'\ th2''\ msg' \Rightarrow$
$(if\ \ (th2'' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)])))$

definition
[*simp*]:
$sc\text{-}cases\text{-}IPC\text{-}SEND\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\ (*check\ if\ th'\ is\ caller\ of\ sc'\ and$

th *is his partner and* msg msg' *are equal*$*$)

then $\{\}$
else $((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \trianglelefteq msg \trianglelefteq th) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \trianglerighteq msg \trianglerighteq th')))$
$|$*P4-IPC-RECV-call* $th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
then $\{\}$
else $(th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$(th' \trianglelefteq msg \trianglelefteq th) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$(th \trianglerighteq msg \trianglerighteq th'))$
$|$*P4-IPC-BUF-call* $th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
then $\{\}$
else $(((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)])))$
$|$*P4-IPC-BUF-RECV-call* $th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
then $\{\}$
else $(th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|$*P4-IPC-MAP-call* $th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
then $\{\}$
else $((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$

```
          IPC MAP  (RECV th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
          IPC MAP  (SEND th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)])))
 |P4-IPC-MAP-RECV-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else (th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
          IPC MAP  (RECV th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
          IPC MAP  (SEND th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]))
 |- ⇒ {})

definition
[simp]:
  sc-cases-IPC-RECV-call th msg th' sc' =
  (case sc' of  P4-IPC-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg') (*check if th' is caller of sc' and
                                                     th is his partner and msg msg' are equal*)
    then {}
    else ((th' ▷ msg ▷ th)
          ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
          (th⊴ msg ⊴ th') ∪
          (th ◁ msg ◁ th')
          ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
          (th' ⊵ msg ⊵ th)))
  |P4-IPC-SEND-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
    then {}
    else ((th' ▷ msg ▷ th)
          ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
          (th⊴ msg ⊴ th') ∪
          (th ◁ msg ◁ th')
          ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
          (th' ⊵ msg ⊵ th)))
  |P4-IPC-BUF-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
    then {}
    else ((
```

$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}BUF\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\ then\ \{\}$
$\ else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ IPC\ BUF\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ IPC\ BUF\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\ then\ \{\}$
$\ else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$\ then\ \{\}$
$\ else\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ IPC\ DONE\ (RECV\ th'\ th\ msg)]))$

|- ⇒ {})

definition
[*simp*]:
sc-cases-IPC-BUF-call th msg th′ sc′ =
(*case sc′ of P4-IPC-call th1′ th2′ msg′* ⇒
(*if* (*th2′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*) (∗*check if th′ is caller of sc′ and th is his partner and msg msg′ are equal*∗)
then {}
else (((*th* ▷ *msg* ▷ *th′*)
⌊[*IPC WAIT* (*SEND th th msg*) , *IPC WAIT* (*RECV th′ th msg*)]⌋
([*IPC PREP* (*RECV th′ th msg*), *IPC WAIT* (*RECV th′ th msg*),
IPC BUF (*RECV th′ th msg*), *IPC DONE* (*SEND th′ th msg*),
IPC DONE (*RECV th′ th msg*)]) ∪
(*th′* ◁ *msg* ◁ *th*)
⌊[*IPC WAIT* (*RECV th′ th msg*) , *IPC WAIT* (*SEND th th′ msg*)]⌋
([*IPC PREP* (*SEND th th′ msg*), *IPC WAIT* (*SEND th th′ msg*),
IPC BUF (*SEND th th′ msg*), *IPC DONE* (*SEND th th′ msg*),
IPC DONE (*RECV th th′ msg*)]) ∪
(*th′* ▷ *msg* ▷ *th*)
⌊[*IPC WAIT* (*SEND th′ th msg*) , *IPC WAIT* (*RECV th th′ msg*)]⌋
([*IPC PREP* (*RECV th th′ msg*), *IPC WAIT* (*RECV th th′ msg*),
IPC BUF (*RECV th th′ msg*), *IPC DONE* (*SEND th th′ msg*),
IPC DONE (*RECV th th′ msg*)]) ∪
(*th* ◁ *msg* ◁ *th′*)
⌊[*IPC WAIT* (*RECV th th′ msg*) , *IPC WAIT* (*SEND th′ th msg*)]⌋
([*IPC PREP* (*SEND th′ th msg*), *IPC WAIT* (*SEND th′ th msg*),
IPC BUF (*SEND th′ th msg*), *IPC DONE* (*SEND th′ th msg*),
IPC DONE (*RECV th′ th msg*)])))))
|*P4-IPC-SEND-call th1′ th2′ msg′* ⇒
(*if* (*th2′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*)
then {}
else ((*th′* ▷ *msg* ▷ *th*)
⌊[*IPC WAIT* (*SEND th′ th msg*) , *IPC WAIT* (*RECV th th′ msg*)]⌋
([*IPC PREP* (*RECV th th′ msg*), *IPC WAIT* (*RECV th th′ msg*),
IPC BUF (*RECV th th′ msg*), *IPC DONE* (*SEND th th′ msg*),
IPC DONE (*RECV th th′ msg*)]) ∪
(*th* ◁ *msg* ◁ *th′*)
⌊[*IPC WAIT* (*RECV th th′ msg*) , *IPC WAIT* (*SEND th′ th msg*)]⌋
([*IPC PREP* (*SEND th′ th msg*), *IPC WAIT* (*SEND th′ th msg*),
IPC BUF (*SEND th′ th msg*), *IPC DONE* (*SEND th′ th msg*),
IPC DONE (*RECV th′ th msg*)])))
|*P4-IPC-RECV-call th1′ th2′ msg′* ⇒
(*if* (*th2′* ≠ *th*) ∨ (*th1′* ≠ *th′*) ∨ (*msg* ≠ *msg′*)
then {}
else (*th* ▷ *msg* ▷ *th′*)
⌊[*IPC WAIT* (*SEND th th msg*) , *IPC WAIT* (*RECV th′ th msg*)]⌋
([*IPC PREP* (*RECV th′ th msg*), *IPC WAIT* (*RECV th′ th msg*),

$IPC\ BUF\ \ (RECV\ th'\ th\ msg)$, $IPC\ DONE\ (SEND\ th'\ th\ msg)$,
$IPC\ DONE\ (RECV\ th'\ th\ msg)$]) $\cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
([$IPC\ PREP\ (SEND\ th\ th'\ msg)$, $IPC\ WAIT\ (SEND\ th\ th'\ msg)$,
$IPC\ BUF\ \ (SEND\ th\ th'\ msg)$, $IPC\ DONE\ (SEND\ th\ th'\ msg)$,
$IPC\ DONE\ (RECV\ th\ th'\ msg)$]))
|*P4-IPC-BUF-call th1′ th2′ msg′* $\Rightarrow$
(*if* $(th2' \neq th) \vee (th1' \neq th') \vee\ (msg \neq msg')$
then {}
else (((*th* $\rhd$ *msg* $\rhd$ *th′*)
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
([$IPC\ PREP\ (RECV\ th'\ th\ msg)$, $IPC\ WAIT\ (RECV\ th'\ th\ msg)$,
$IPC\ BUF\ \ (RECV\ th'\ th\ msg)$, $IPC\ DONE\ (SEND\ th'\ th\ msg)$,
$IPC\ DONE\ (RECV\ th'\ th\ msg)$]) $\cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
([$IPC\ PREP\ (SEND\ th\ th'\ msg)$, $IPC\ WAIT\ (SEND\ th\ th'\ msg)$,
$IPC\ BUF\ \ (SEND\ th\ th'\ msg)$, $IPC\ DONE\ (SEND\ th\ th'\ msg)$,
$IPC\ DONE\ (RECV\ th\ th'\ msg)$]) $\cup$
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
([$IPC\ PREP\ (RECV\ th\ th'\ msg)$, $IPC\ WAIT\ (RECV\ th\ th'\ msg)$,
$IPC\ BUF\ \ (RECV\ th\ th'\ msg)$, $IPC\ DONE\ (SEND\ th\ th'\ msg)$,
$IPC\ DONE\ (RECV\ th\ th'\ msg)$]) $\cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
([$IPC\ PREP\ (SEND\ th'\ th\ msg)$, $IPC\ WAIT\ (SEND\ th'\ th\ msg)$,
$IPC\ BUF\ \ (SEND\ th'\ th\ msg)$, $IPC\ DONE\ (SEND\ th'\ th\ msg)$,
$IPC\ DONE\ (RECV\ th'\ th\ msg)$])))))
|*P4-IPC-BUF-SEND-call th1′ th2′ msg′* $\Rightarrow$
(*if* $(th2' \neq th) \vee (th1' \neq th') \vee\ (msg \neq msg')$
then {}
else $(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
([$IPC\ PREP\ (RECV\ th\ th'\ msg)$, $IPC\ WAIT\ (RECV\ th\ th'\ msg)$,
$IPC\ BUF\ \ (RECV\ th\ th'\ msg)$, $IPC\ DONE\ (SEND\ th\ th'\ msg)$,
$IPC\ DONE\ (RECV\ th\ th'\ msg)$]) $\cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
([$IPC\ PREP\ (SEND\ th'\ th\ msg)$, $IPC\ WAIT\ (SEND\ th'\ th\ msg)$,
$IPC\ BUF\ \ (SEND\ th'\ th\ msg)$, $IPC\ DONE\ (SEND\ th'\ th\ msg)$,
$IPC\ DONE\ (RECV\ th'\ th\ msg)$]))
|*P4-IPC-BUF-RECV-call th1′ th2′ msg′* $\Rightarrow$
(*if* $(th2' \neq th) \vee (th1' \neq th') \vee\ (msg \neq msg')$
then {}
else $(th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
([$IPC\ PREP\ (RECV\ th'\ th\ msg)$, $IPC\ WAIT\ (RECV\ th'\ th\ msg)$,

```
          IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
          IPC BUF  (SEND th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]))
  |- ⇒ {})

definition
[simp]:
  sc-cases-IPC-BUF-SEND-call th msg th' sc' =
  (case sc' of  P4-IPC-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg') (*check if th' is caller of sc' and
                                                     th is his partner and msg msg' are equal*)
    then {}
    else (((th ▷ msg ▷ th')
          ⌊[IPC WAIT (SEND th th msg), IPC WAIT (RECV th' th msg) ]⌋
          ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
            IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
            IPC DONE (RECV th' th msg)]) ∪
          (th' ◁ msg ◁ th)
          ⌊[IPC WAIT (RECV th' th msg), IPC WAIT (SEND th th' msg)]⌋
          ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
            IPC BUF  (SEND th th' msg), IPC DONE (SEND th th' msg),
            IPC DONE (RECV th th' msg)])))))
  |P4-IPC-RECV-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
    then {}
    else (th ▷ msg ▷ th')
          ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
          ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
            IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
            IPC DONE (RECV th' th msg)]) ∪
          (th' ◁ msg ◁ th)
          ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
          ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
            IPC BUF  (SEND th th' msg), IPC DONE (SEND th th' msg),
            IPC DONE (RECV th th' msg)]))
  |P4-IPC-BUF-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
    then {}
    else (((th ▷ msg ▷ th')
          ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
          ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
            IPC BUF  (RECV th' th msg), IPC DONE (SEND th' th msg),
            IPC DONE (RECV th' th msg)]) ∪
          (th' ◁ msg ◁ th)
```

$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)])))$
$|P4\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ (th2' \neq th) \vee (th1' \neq th') \vee (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ BUF\ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ BUF\ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|- \Rightarrow \{\})$

definition
[*simp*]:
$sc\text{-}cases\text{-}IPC\text{-}BUF\text{-}RECV\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ (th2' \neq th) \vee (th1' \neq th') \vee (msg \neq msg')\ (*check\ if\ th'\ is\ caller\ of\ sc'\ and$

$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ ((($
$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ BUF\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ (th2' \neq th) \vee (th1' \neq th') \vee (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ BUF\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$

```
        IPC BUF  (SEND th' th msg), IPC DONE (SEND th' th msg),
        IPC DONE (RECV th' th msg)])))
 |P4-IPC-BUF-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else ((
        (th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
          IPC BUF  (RECV th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
          IPC BUF  (SEND th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)])))))
 |P4-IPC-BUF-SEND-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else (th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
          IPC BUF  (RECV th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
          IPC BUF  (SEND th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]))
 |- ⇒ {})
```

definition

```
[simp]:
  sc-cases-IPC-MAP-call th msg th' sc' =
  (case sc' of  P4-IPC-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg') (*check if th' is caller of sc' and

                                                    th is his partner and msg msg' are equal*)
    then {}
    else (((th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
          IPC MAP  (RECV th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
          IPC MAP  (SEND th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]) ∪
```

$(th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]))$
$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (((th \rhd msg \rhd th')$
$\lfloor[IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),$
$IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup$
$(th' \lhd msg \lhd th)$
$\lfloor[IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),$
$IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$

```
        (th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
          IPC MAP  (RECV th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
          IPC MAP  (SEND th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)])))
 |P4-IPC-MAP-SEND-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else (th' ▷ msg ▷ th)
        ⌊[IPC WAIT (SEND th' th msg) , IPC WAIT (RECV th th' msg) ]⌋
        ([IPC PREP (RECV th th' msg), IPC WAIT (RECV th th' msg),
          IPC MAP  (RECV th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]) ∪
        (th ◁ msg ◁ th')
        ⌊[IPC WAIT (RECV th th' msg) , IPC WAIT (SEND th' th msg)]⌋
        ([IPC PREP (SEND th' th msg), IPC WAIT (SEND th' th msg),
          IPC MAP  (SEND th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]))
 |P4-IPC-MAP-RECV-call th1' th2' msg' ⇒
  (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg')
   then {}
   else (th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg) , IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
          IPC MAP  (RECV th' th msg), IPC DONE (SEND th' th msg),
          IPC DONE (RECV th' th msg)]) ∪
        (th' ◁ msg ◁ th)
        ⌊[IPC WAIT (RECV th' th msg) , IPC WAIT (SEND th th' msg)]⌋
        ([IPC PREP (SEND th th' msg), IPC WAIT (SEND th th' msg),
          IPC MAP  (SEND th th' msg), IPC DONE (SEND th th' msg),
          IPC DONE (RECV th th' msg)]))
 |- ⇒ {})
```

definition

```
[simp]:
  sc-cases-IPC-MAP-SEND-call th msg th' sc' =
  (case sc' of  P4-IPC-call th1' th2' msg' ⇒
   (if  (th2' ≠ th) ∨ (th1' ≠ th') ∨  (msg ≠ msg') (*check if th' is caller of sc' and

                                                    th is his partner and msg msg' are equal*)
    then {}
    else (((th ▷ msg ▷ th')
        ⌊[IPC WAIT (SEND th th msg), IPC WAIT (RECV th' th msg) ]⌋
        ([IPC PREP (RECV th' th msg), IPC WAIT (RECV th' th msg),
```

$$
\begin{array}{l}
\quad\quad IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup\\
\quad\quad (th' \lhd msg \lhd th)\\
\quad\quad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor\\
\quad\quad ([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th\ th'\ msg)])))\\
|P4\text{-}IPC\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow\\
\ (if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\\
\ \ then\ \{\}\\
\ \ else\ (th \rhd msg \rhd th')\\
\quad\quad \lfloor [IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ] \rfloor\\
\quad\quad ([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),\\
\quad\quad IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup\\
\quad\quad (th' \lhd msg \lhd th)\\
\quad\quad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor\\
\quad\quad ([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th\ th'\ msg)]))\\
|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow\\
\ (if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\\
\ \ then\ \{\}\\
\ \ else\ (((th \rhd msg \rhd th')\\
\quad\quad \lfloor [IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ] \rfloor\\
\quad\quad ([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),\\
\quad\quad IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup\\
\quad\quad (th' \lhd msg \lhd th)\\
\quad\quad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor\\
\quad\quad ([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th\ th'\ msg)])))\\
|P4\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th1'\ th2'\ msg' \Rightarrow\\
\ (if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\\
\ \ then\ \{\}\\
\ \ else\ (th \rhd msg \rhd th')\\
\quad\quad \lfloor [IPC\ WAIT\ (SEND\ th\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ] \rfloor\\
\quad\quad ([IPC\ PREP\ (RECV\ th'\ th\ msg),\ IPC\ WAIT\ (RECV\ th'\ th\ msg),\\
\quad\quad IPC\ MAP\ \ (RECV\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th'\ th\ msg)]) \cup\\
\quad\quad (th' \lhd msg \lhd th)\\
\quad\quad \lfloor [IPC\ WAIT\ (RECV\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (SEND\ th\ th'\ msg)] \rfloor\\
\quad\quad ([IPC\ PREP\ (SEND\ th\ th'\ msg),\ IPC\ WAIT\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ MAP\ \ (SEND\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),\\
\quad\quad IPC\ DONE\ (RECV\ th\ th'\ msg)]))\\
|\text{-} \Rightarrow \{\})
$$

definition

$[simp]$:
$sc\text{-}cases\text{-}IPC\text{-}MAP\text{-}RECV\text{-}call\ th\ msg\ th'\ sc' =$
$(case\ sc'\ of\ \ P4\text{-}IPC\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')\ (*check\ if\ th'\ is\ caller\ of\ sc'\ and$

$th\ is\ his\ partner\ and\ msg\ msg'\ are\ equal*)$
$then\ \{\}$
$else\ ((\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ \ IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ \ IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)])))$
$|P4\text{-}IPC\text{-}MAP\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ ((\ (th' \rhd msg \rhd th)$
$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$\ \ IPC\ MAP\ \ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$\ \ IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ MAP\ \ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$\ \ IPC\ DONE\ (RECV\ th'\ th\ msg)])))))$
$|P4\text{-}IPC\text{-}MAP\text{-}SEND\text{-}call\ th1'\ th2'\ msg' \Rightarrow$
$(if\ \ (th2' \neq th) \vee (th1' \neq th') \vee\ \ (msg \neq msg')$
$then\ \{\}$
$else\ (th' \rhd msg \rhd th)$

$\lfloor[IPC\ WAIT\ (SEND\ th'\ th\ msg)\ ,\ IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ]\rfloor$
$([IPC\ PREP\ (RECV\ th\ th'\ msg),\ IPC\ WAIT\ (RECV\ th\ th'\ msg),$
$IPC\ MAP\ (RECV\ th\ th'\ msg),\ IPC\ DONE\ (SEND\ th\ th'\ msg),$
$IPC\ DONE\ (RECV\ th\ th'\ msg)]) \cup$
$(th \lhd msg \lhd th')$
$\lfloor[IPC\ WAIT\ (RECV\ th\ th'\ msg)\ ,\ IPC\ WAIT\ (SEND\ th'\ th\ msg)]\rfloor$
$([IPC\ PREP\ (SEND\ th'\ th\ msg),\ IPC\ WAIT\ (SEND\ th'\ th\ msg),$
$IPC\ MAP\ (SEND\ th'\ th\ msg),\ IPC\ DONE\ (SEND\ th'\ th\ msg),$
$IPC\ DONE\ (RECV\ th'\ th\ msg)]))$
$|\text{-} \Rightarrow \{\})$

definition

[*simp*]:

comm-cases th th' sc sc' =
(case sc of P4-IPC-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th') (∗check if th is caller of sc and th' is his partner∗)
then {}
else sc-cases-IPC-call th msg th' sc')
|P4-IPC-SEND-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-SEND-call th msg th' sc')
|P4-IPC-RECV-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-RECV-call th msg th' sc')
|P4-IPC-BUF-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-BUF-call th msg th' sc')
|P4-IPC-BUF-SEND-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-BUF-SEND-call th msg th' sc')
|P4-IPC-BUF-RECV-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-BUF-RECV-call th msg th' sc')
|P4-IPC-MAP-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-MAP-call th msg th' sc')
|P4-IPC-MAP-SEND-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}
else sc-cases-IPC-MAP-SEND-call th msg th' sc')
|P4-IPC-MAP-RECV-call th1' th2' msg ⇒
(if (th2' ≠ th') ∨ (th1' ≠ th) ∨ (th = th')
then {}

else sc-cases-IPC-MAP-RECV-call th msg th′ sc′))

fun *criteria* :: (*′th-id*, (*′th-id*, *′msg*) *P4-IPC-call*)*criterion* ⇒
((*p4-stage*$_{ipc}$, (*′th-id*, *′msg*) *p4-direct*$_{ipc}$)*action*$_{ipc}$ *list*) *set*
where
criteria (*interleave-all S*) = *undefined*

|*criteria* (*COMM th th′ scTab*) =
(*case scTab th of None* ⇒ {}
| *Some sc* ⇒
(*case scTab th′ of None* ⇒ {}
| *Some sc′* ⇒ *comm-cases th th′ sc sc′*))

|*criteria* (*TPAIR th th′ scTab*) =
(*case scTab th of None* ⇒
(*case scTab th′ of None* ⇒ {}
| *Some sc* ⇒
{*IPC-call-sem sc*})
| *Some sc* ⇒
(*case scTab th′ of None* ⇒ {*IPC-call-sem sc*}
| *Some sc′* ⇒ *interleave* (*IPC-call-sem sc*) (*IPC-call-sem sc′*)))

Q.5 Partial order theorem

lemma *partial-order-ipc-instance-resource*:
assumes *1*: *th* ≠ *th′*
shows
image (λ *is. mbind is* (λ*a.* (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$ *a*))
σ)
(*criteria* (*COMM th th′* [*th* ↦ *P4-IPC-call th th′ msg* ,
th′ ↦ *P4-IPC-call th′ th msg*])) =
image (λ *is. mbind is* (λ*a.* (*out1* ← *BUF-RECV*$_{MON}$ *a* ; *MAP-RECV*$_{MON}$
a)) *σ*)
(*interleave* (*th* ◁ *msg* ◁ *th′*) (*th′* ▷ *msg* ▷ *th*))
oops

lemma (*int o card*) (*criteria* (*COMM th th′* [*th* ↦ *P4-IPC-call th th′ msg* ,
th′ ↦ *P4-IPC-call th′ th msg*])) <
(*int o card*) ((*interleave* (*th* ◁ *msg* ◁ *th′*) (*th′* ▷ *msg* ▷ *th*)))
by *simp*

Q.6 ipc communications derivations

Q.7 Lemmas on ipc communications

lemma *comm-with-P4-IPC-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th′ msg*) ∧
(*the o scTab*) *th′* = (*P4-IPC-call th′ th msg*)

```
    and 2: th ∈ dom scTab ∧ th' ∈ dom scTab
    and 3: th ≠ th'
 shows      criteria (COMM th th' scTab) ≠ {}
proof (cases scTab th)
  fix scTab th
  case None
  from this
  show ?thesis
  using assms
  by auto
next
  case (Some a)
  from this
  show ?thesis
  using assms
  by auto
qed

lemma comm-with-P4-IPC-BUF-call-Some:
 assumes 1:(the o scTab) th = (P4-IPC-call th th' msg) ∧
          (the o scTab) th' = (P4-IPC-BUF-call th' th msg)
    and 2: th ∈ dom scTab ∧ th' ∈ dom scTab
    and 3: th ≠ th'
 shows      criteria (COMM th th' scTab) ≠ {}
proof (cases scTab th)
  case None
  assume 1: scTab th = None
  then show ?thesis
  using assms
  by auto
next
  case (Some a)
  assume 1: scTab th = Some a
  then show ?thesis
  using assms
  by (auto simp: split:option.split )
qed

lemma comm-with-P4-IPC-BUF-SEND-call-Some:
 assumes 1:(the o scTab) th = (P4-IPC-call th th' msg) ∧
          (the o scTab) th' = (P4-IPC-BUF-SEND-call th' th msg)
    and 2: th ∈ dom scTab ∧ th' ∈ dom scTab
    and 3: th ≠ th'
 shows      criteria (COMM th th' scTab) ≠ {}
proof (cases scTab th)
  case None
  assume 1: scTab th = None
  then show ?thesis
  using assms
```

by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-BUF-RECV-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-BUF-RECV-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-MAP-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-MAP-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-MAP-SEND-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-MAP-SEND-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *comm-with-P4-IPC-MAP-RECV-call-Some*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-MAP-RECV-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) ≠ {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

Q.8 No communications

lemma *not-comm-SEND-SEND*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-SEND-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-SEND-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)

case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-SEND-SEND-BUF*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-SEND-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-BUF-SEND-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-SEND-SEND-MAP*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-SEND-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-MAP-SEND-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*

using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-RECV-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV-BUF*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-BUF-RECV-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*
shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

lemma *not-comm-RECV-RECV-MAP*:
assumes *1*:(*the o scTab*) *th* = (*P4-IPC-RECV-call th th' msg*) ∧
(*the o scTab*) *th'* = (*P4-IPC-MAP-RECV-call th' th msg*)
and *2*: *th* ∈ *dom scTab* ∧ *th'* ∈ *dom scTab*
and *3*: *th* ≠ *th'*

shows *criteria* (*COMM th th' scTab*) = {}
proof (*cases scTab th*)
case *None*
assume *1*: *scTab th* = *None*
then show *?thesis*
using *assms*
by *auto*
next
case (*Some a*)
assume *1*: *scTab th* = *Some a*
then show *?thesis*
using *assms*
by (*auto simp*: *split*:*option.split*)
qed

end

Bibliography

[ABC+13] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[ABGR10] Wolfgang Ahrendt, Bernhard Beckert, Martin Giese, and Philipp Rümmer. Practical aspects of automated deduction for program verification. *KI*, 24(1):43–49, 2010.

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings.* Cambridge University Press, New York, NY, USA, 1996.

[AK95] D.P. Appenzeller and A. Kuehlmann. Formal verification of a powerpc microprocessor. In *Computer Design: VLSI in Computers and Processors, 1995. ICCD '95. Proceedings., 1995 IEEE International Conference on*, pages 79–84, oct 1995.

[All70a] Frances E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[All70b] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970.

[And86] Peter B. Andrews. *An introduction to mathematical logic and type theory: to truth through proof.* Academic Press Professional, Inc., San Diego, CA, USA, 1986.

[And02] Peter B. Andrews. *Introduction to Mathematical Logic and Type Theory: To Truth through Proof.* Kluwer Academic Publishers, Dordrecht, 2002.

[Bal10] Clemens Ballarin. *Tutorial to Locales and Locale Interpretation*, 2010.

[Bar03] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[BBW15] Achim D. Brucker, Lukas Brügger, and Burkhart Wolff. Formal firewall conformance testing: an application of test and proof techniques. *Softw. Test., Verif. Reliab.*, 25(1):34–71, 2015.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Proceedings of the 23rd International Conference on Computer Aided Verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BCFG86] L Bougé, N Choquet, L Fribourg, and M C Gaudel. Test sets generation from algebraic specifications using logic programming. *J. Syst. Softw.*, 6(4):343–360, Nov 1986.

[BFNW13] Achim D. Brucker, Abderrahmane Feliachi, Yakoub Nemouchi, and Burkhart Wolff. Test program generation for a microprocessor. *Lecture Notes in Computer Science*, 7942:76–95, 2013.

[BFY+97] P. Biswas, A. Freeman, K. Yamada, N. Nakagawa, and K. Uchiyama. Functional verification of the superscalar sh-4 microprocessor. In *Compcon '97. Proceedings, IEEE*, pages 115–120, feb 1997.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Softw. Eng. J.*, 6(6):387–405, Nov 1991.

[BHJJ08] Gregorv. Bochmann, Stefan Haar, Claude Jard, and Guy-Vincent Jourdan. Testing systems specified as partial order input/output automata. In Kenji Suzuki, Teruo Higashino, Andreas Ulrich, and Toru Hasegawa, editors, *Testing of Software and Communicating Systems*, volume 5047 of *Lecture Notes in Computer Science*, pages 169–183. Springer Berlin Heidelberg, 2008.

[BHNW15a] Achim D. Brucker, Oto Havle, Yakoub Nemouchi, and Burkhart Wolff. Testing the IPC protocol for a real-time operating system. In *Verified Software: Theories, Tools, and Experiments - 7th International Conference, VSTTE 2015, San Francisco, CA, USA, July 18-19, 2015. Revised Selected Papers*, pages 40–60, 2015.

[BHNW15b] Achim D. Brucker, Oto Havle, Yakoub Nemouchi, and Burkhart Wolff. Testing the IPC protocol for a real-time operating system. In Arie Gurfinkel and Sanjit A. Seshia, editors, *Working Conference on Verified Software: Theories, Tools, and Experiments*, Lecture Notes in Computer Science. Springer-Verlag, Heidelberg, 2015.

[BJK+06] Sven Beyer, Christian Jacobi, Daniel Kröning, Dirk Leinenbach, and Wolfgang J. Paul. Putting it all together – Formal verification of the VAMP. *International Journal on Software Tools for Technology Transfer*, 8(4):411–430, 2006.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated Testing Based on Java Predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[BP95] Bernhard Beckert and Joachim Posegga. leanTAP: Lean tableau-based deduction. *Journal of Automated Reasoning*, 15(3):339–358, 1995.

[Bro11] Benjamin Brosgol. Do-178c: The next avionics safety standard. In *Proceedings of the 2011 ACM Annual International Conference on Special Interest Group on the Ada Programming Language*, SIGAda '11, pages 5–6, New York, NY, USA, 2011. ACM.

[BW07] Achim D. Brucker and Burkhart Wolff. Test-sequence generation with hol-testgen with an application to firewall testing. In *Tests and Proofs*, pages 149–168, 2007.

[BW09] Achim D. Brucker and Burkhart Wolff. HOL-TestGen: An interactive test-case generation framework. In Marsha Chechik and Martin Wirsing, editors, *Fundamental Approaches to Software Engineering (FASE)*, number 5503 in Lecture Notes in Computer Science, pages 417–420. Springer-Verlag, 2009.

[BW10] Sascha Böhme and Tjark Weber. Fast LCF-Style Proof Reconstruction for Z3. In *ITP*, pages 179–194, 2010.

[BW13] Achim D. Brucker and Burkhart Wolff. On Theorem Prover-based Testing. *Formal Asp. Comput. (FAOC)*, 25(5):683–721, 2013.

[Car81] Robert Cartwright. Formal program testing. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of*

Programming Languages, POPL '81, pages 125–132, New York, NY, USA, 1981. ACM.

[CDH+09] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichalĆ Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A Practical System for Verifying Concurrent C. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer Berlin Heidelberg, 2009.

[CG07] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in CSP. In *Proceedings of the Formal Engineering Methods 9th International Conference on Formal Methods and Software Engineering*, pages 151–170, Berlin, Heidelberg, 2007. Springer-Verlag.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, pages 56–68, June 1940.

[CKK+12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: a software analysis perspective. In *International Conference on Software Engineering and Formal Methods (SEFM'12)*, pages 233–247. Springer, October 2012.

[Com06] Common criteria. *Common criteria for information technology security evaluation ((version 3.1), Part: Security assurance components*, September 2006.

[CSCS94] David Carrington, Phil Stocks, D. Carrington, and P. Stocks. A tale of two paradigms: Formal methods and software testing. 1994.

[DGM93] P. Dauchy, M.-C. Gaudel, and B. Marre. Using algebraic specifications in software testing: A case study on the software of an automatic subway. *J. Syst. Softw.*, 21(3), Jun 1993.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *TACAS*, 2008.

[Dor10] Jan Dorrenbacher. *Formal Specification and Verification of Microkernel*. PhD thesis, Saarland University, Saarbrücken, Germany, 2010.

[DY96] D.Lee and M. Yannakakis. Principles and method of testing finite state machines- a survey. In *Proceedings of the IEEE*, volume 84, pages 1090–1123. IEEE, Aug 1996.

[EFH83] Hartmut Ehrig, Werner Fey, and Horst Hansen. ACT ONE - an algebraic specification language with two levels of semantics. In *ADT*, 1983.

[EH08] Abdeslam En-Nouaary and Abdelwahab Hamou-Lhadj. A boundary checking technique for testing real-time systems modeled as timed input output automata (short paper). In *Proceedings of the Eighth International Conference on Quality Software, QSIC 2008, 12-13 August 2008, Oxford, UK*, pages 209–215, 2008.

[En-13] Abdeslam En-Nouaary. A test purpose-based approach for testing timed input output automata. *Softw. Test., Verif. Reliab.*, 23(1):53–76, 2013.

[Fel12] Abderrahmane Feliachi. *Semantics-Based Testing for Circus.* Theses, Université Paris Sud - Paris XI, December 2012.

[FGWW13] Abderrahmane Feliachi, Marie-Claude Gaudel, Makarius Wenzel, and Burkhart Wolff. The circus testing theory revisited in Isabelle/HOL. In *Formal Methods and Software Engineering*, pages 131–147, 2013.

[Fox03] Anthony C. J. Fox. Formal specification and verification of arm6. In *TPHOLs*, pages 25–40, 2003.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, March 2013.

[FT01] F. Fallah and K. Takayama. A new functional test program generation methodology. In *Computer Design, 2001. ICCD 2001. Proceedings. 2001 International Conference on*, pages 76–81, 2001.

[FTW04] Lars Frantzen, Jan Tretmans, and Tim A. C. Willemse. Test generation based on symbolic specifications. In *FATES*, pages 1–15, 2004.

[GAK12] David Greenaway, June Andronick, and Gerwin Klein. Bridging the gap: Automatic verified abstraction of C. In *ITP*, pages 99–115, 2012.

[Gau95] Marie-Claude Gaudel. Testing can be formal, too. In *Proceedings of the 6th International Joint Conference CAAP/FASE*

on Theory and Practice of Software Development, pages 82–96. Springer-Verlag, 1995.

[Gau10] Marie-Claude Gaudel. Software testing based on formal specification. In Paulo Borba, Ana Cavalcanti, Augusto Sampaio, and Jim Woodcook, editors, *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 215–242. Springer Berlin Heidelberg, 2010.

[GB91] M.-C Gaudel G.Bernot and B.Marre. Software testing based on formal specification: Atheory and a tool. *Software Engeneering Journal*, 6(6):387–405, November 1991.

[Gil62] Arthur Gill. *Introduction to the theory of finite-state machines.* McGraw-Hill, 1962.

[GLG08] Marie-Claude Gaudel and Pascale Le Gall. Testing data types implementations from algebraic specifications. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, pages 209–239. Springer-Verlag, 2008.

[GM93] Mike J. C. Gordon and Tom F. Melham. *Introduction to HOL.* Cambridge University Press, 1993.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, July 1981.

[Gon13] Georges Gonthier. Engineering mathematics: the odd order theorem proof. In *POPL*, pages 1–2, 2013.

[Gor00] Mike Gordon. From lcf to hol: a short history. In *Proof, Language, and Interaction*, pages 169–185. MIT Press, 2000.

[GW94] P. Godefroid and P. Wolper. A partial approach to model checking. In *Papers Presented at the IEEE Symposium on Logic in Computer Science*, number 22, pages 305–326, Orlando, FL, USA, 1994. Academic Press, Inc.

[Haf15] Florian Haftmann. Code generation from Isabelle/HOL theories, May 2015.

[Hal08] Thomas C Hales. Formal proof. *Notices of the AMS*, 55(11):1370–1380, 2008.

[Har03] John Harrison. Formal verification at Intel. In *LICS*, pages 45–. IEEE Computer Society, 2003.

[Har06] John Harrison. Towards self-verification of HOL Light. In *IJCAR*, pages 177–191, 2006.

[Har09] John Harrison. HOL Light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66, Munich, Germany, 2009. Springer-Verlag.

[Hay84] John P. Hayes. Fault modeling for digital MOS integrated circuits. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 3(3):200–208, 1984.

[HB07] Vance Hilderman and Tony Baghai. *Avionics Certification: A Complete Guide to DO-178 (Software), DO-254 (Hardware).* Avionics Communications Inc., 2007.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H. Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):9:1–9:76, February 2009.

[Hen64] F. C. Hennine. Fault detecting experiments for sequential circuits. In *Proceedings of the 1964 Proceedings of the Fifth Annual Symposium on Switching Circuit Theory and Logical Design*, SWCT '64, pages 95–110, Washington, DC, USA, 1964. IEEE Computer Society.

[HP06] John L. Hennessy and David A. Patterson. *Computer Architecture, Fourth Edition: A Quantitative Approach.* Morgan Kaufmann Publishers Inc., 2006.

[Hur03] Joe Hurd. First-order proof tactics in higher-order logic theorem provers. In *Design and Application of Strategies/Tactics in Higher Order Logics, number NASA/CP-2003-212448 in NASA Technical Reports*, pages 56–68, 2003.

[Int12] ISO/IEC DIS 29119: Software and Systems Engineering—Software Testing. ISO Draft International Standard, July 2012.

[JH08] Éric Jaeger and Thérèse Hardin. A few remarks about formal development of secure systems. In *11th IEEE High Assurance Systems Engineering Symposium, HASE 2008, Nanjing, China, December 3 - 5, 2008*, pages 165–174, 2008.

[KAMO14] Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. HOL with definitions: Semantics, soundness, and a verified implementation. In *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, pages 308–324, 2014.

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES*, pages 207–220. ACM, 2009.

[KKV11] Alexander Kamkin, Eugene Kornykhin, and Dmitry Vorobyev. Reconfigurable model-based test program generator for microprocessors. *Software Testing Verification and Validation Workshop, IEEE International Conference on*, 0:47–54, 2011.

[Kum13] Ramana Kumar. Challenges in using opentheory to transport harrison's HOL model from HOL light to HOL4. In Jasmin Christian Blanchette and Josef Urban, editors, *PxTP 2013. Third International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series in Computing*, pages 110–116. EasyChair, 2013.

[LDvB+93] Gang Luo, Rachida Dssouli, Gregor von Bochmann, Pallapa Venkataram, and Abderrazak Ghedamsi. Generating synchronizable test sequences based on finite state machine with distributed ports. In *Protocol Test Systems, VI, Proceedings of the IFIP TC6/WG6.1 Sixth International Workshop on Protocol Test systems, Pau, France, 28-30 September, 1993*, pages 139–153, 1993.

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[Lie95] Liedtke. on ţ-kernel construction. *15th SOSP, Copper Mountain, CO, USA*, pages 237–250, December 1995.

[LM96] D. Lee and M.Yannakakis. Principles and method of testing finite state machines- a survey. *Proceeding of the IEEE*, 84(8):1090–1126, 1996.

[LT89a] Nancy Lynch and Mark Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[LT89b] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2:219–246, 1989.

[LY94] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, Mar 1994.

[MD08] Prabhat Mishra and Nikil Dutt. Specification-driven directed test generation for validation of pipelined processors. *ACM Trans. Design Autom. Electr. Syst.*, 13(3), 2008.

[Mem06] The Common Criteria Recognition Agreement Members. Common criteria for information technology security evaluation. http://www.commoncriteriaportal.org/, Sep 2006.

[Mer01] Stephan Merz. Model checking: A tutorial overview. In *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-verlag, June 2001.

[MHST08] Till Mossakowski, AnneE. Haxthausen, Donald Sannella, and Andrezj Tarlecki. Casl âĂŤ the common algebraic specification language. In Dines BjÃÿrner and MartinC. Henson, editors, *Logics of Specification Languages*, Monographs in Theoretical Computer Science, pages 241–298. Springer Berlin Heidelberg, 2008.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348 – 375, 1978.

[MOK13] Magnus O. Myreen, Scott Owens, and Ramana Kumar. Steps towards verified implementations of HOL Light. In *ITP*, pages 490–495, 2013.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[MQB07] Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. Chess: A systematic testing tool for concurrent software. Technical Report MSR-TR-2007-149, Microsoft Research, November 2007.

[MTM97] Robin Milner, Mads Tofte, and David Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[MW79] R. Milner and C.P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation.* Lecture Notes in Computer Science. Springer, 1979.

[MW10] David C. J. Matthews and Makarius Wenzel. Efficient parallel programming in poly/ml and isabelle/ml. In *Proceedings of the POPL 2010 Workshop on Declarative Aspects of Multicore Programming, DAMP 2010, Madrid, Spain, January 19, 2010*, pages 53–62, 2010.

[Nip12] Tobias Nipkow. Theory fun, 2012.

[NPW02] Tobias Nipkow, Larry C. Paulson, and Markus Wenzel. *Isabelle/**hol!**—A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, Heidelberg, 2002.

[NWS+] Wolfgang Naraschewski, Markus Wenzel, Norbert Schirmer, Thomas Sewell, and Florian Haftmann. Theory record.

[Pau99] Lawrence C. Paulson. A generic tableau prover and its integration with isabelle. *J. UCS*, 5(3):73–87, 1999.

[Pel93] Doron Peled. All from one, one for all: on model checking using representatives. In *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, pages 409–423, 1993.

[PHL12] Hernán Ponce de León, Stefan Haar, and Delphine Longuet. Conformance relations for labeled event structures. In *Tests and Proofs - 6th International Conference, TAP 2012, Prague, Czech Republic, May 31 - June 1, 2012. Proceedings*, volume 7305, pages 83–98, 2012.

[PS83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Comput. Surv.*, 15(3):199–236, September 1983.

[RJB99] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language Reference Manual.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1999.

[Rus99] David M. Russinoff. A mechanically checked proof of correctness of the amd k5 floating point square root microcode. *Formal Methods in System Design*, 14(1):75–125, 1999.

[SLZ07] Weihang Jiang Shan Lu and Yuanyuan Zhou. A study of interleaving coverage criteria. In *The 6th Joint Meeting on*

European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers, pages 533–536, September 2007.

[SMZ05] Haihua Shen, Lin Ma, and Heng Zhang. Crpg: a configurable random test-program generator for microprocessors. In *Circuits and Systems, 2005. ISCAS 2005. IEEE International Symposium on*, pages 4171–4174 Vol. 4, may 2005.

[SR10] Heiko Stallbaum and Mark Rzepka. Toward do-178b-compliant test models. In *Proceedings of the 7th Workshop on Model-Driven Engineering, Verification and Validation, MoDeVVa 2010, at 13th Intl. Conference on Model Driven Engineering Languages and Systems, Models 2010 (Oslo, Norway, 3rd of October 2010)*, October 2010.

[SV03] Sudarshan K. Srinivasan and Miroslav N. Velev. Formal verification of an intel xscale processor model with scoreboarding, specialized execution pipelines, and impress data-memory exceptions. In *MEMOCODE*, pages 65–74. IEEE Computer Society, 2003.

[SYS13a] SYSGO. *PikeOS Fundamentals*. SYSGO, 2013.

[SYS13b] SYSGO. *PikeOS Kernel*. SYSGO, 2013.

[TPHS10] Jan Tretmans, Florian Prester, Philipp Helle, and Wladimir Schamai. Model-based testing 2010: Short abstracts. *Electr. Notes Theor. Comput. Sci.*, 264(3):85–99, 2010.

[Tre08a] Jan Tretmans. Formal methods and testing. chapter Model Based Testing with Labelled Transition Systems, pages 1–38. Springer-Verlag, Berlin, Heidelberg, 2008.

[Tre08b] Jan Tretmans. Model based testing with labelled transition systems. In Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors, *Formal Methods and Testing*, volume 4949, pages 1–38. Springer-Verlag, 2008.

[Urb] Christian Urban. The isabelle cookbook: A gentle tutorial for programming on the ml-level of isabelle, July.

[Wad92] Philip Wadler. Comprehending monads. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 61–78, New York, NY, USA, 1992.

[WB89] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76, 1989.

[Wen97] Markus Wenzel. Type classes and overloading in higher-order logic. In *TPHOLs*, pages 307–322, 1997.

[Wen02] Markus M Wenzel. *Isabelle/Isar—a versatile environment for human-readable formal proof documents*. PhD thesis, Technische Universität München, Universitätsbibliothek, 2002.

[Wen15] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, May 2015.

[Wie06] Freek Wiedijk. *The Seventeen Provers of the World: Foreword by Dana S. Scott (Lecture Notes in Computer Science / Lecture Notes in Artificial Intelligence)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[YABC15] Y.Nemouchi, A.Feliachi, B.Wolff, and C.Proch. Isabelle in certification processes, Dec 2015.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys (CSUR)*, 29(4):366–427, 1997.