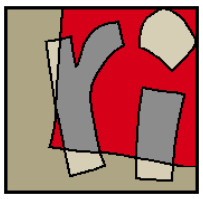
# Semantics-Based Testing for Circus

Abderrahmane Feliachi


Université Paris-Sud

École Doctorale Informatique Paris-Sud

Laboratoire de Recherche en Informatique

Thèse de Doctorat

---

# Semantics-Based Testing for Circus

---

*Présentée par:*
**Abderrahmane Feliachi**
*pour l'obtention du*
**Doctorat de l'université Paris-Sud XI**

**Jury**

| | | |
|---|---|---|
| Pr. Rob Hierons | Brunel University, UK | Rapporteur |
| Pr. Stephan Merz | INRIA Nancy | Rapporteur |
| Dr. Ana Cavalcanti | University of York, UK | Examinatrice |
| Pr. Pascale Le Gall | Université d'Evry | Examinatrice |
| Pr. Claude Marché | INRIA Saclay | Examinateur |
| Pr. Marie-Claude Gaudel | Université Paris-Sud XI | Directrice de thèse |
| Pr. Burkhart Wolff | Université Paris-Sud XI | Co-Encadrant |

soutenue le 12.12.12

# Acknowledgments

This PhD would not have been possible without the guidance and the help of several individuals who in one way or another contributed and extended their valuable assistance in the preparation and completion of this thesis.

I would like to express my deepest gratitude to my supervisors, Pr. Marie-Claude Gaudel and Pr. Burkhart Wolff, for their excellent guidance, caring, patience, advices, and for offering me the best of their knowledge and experience in scientific research. This thesis would not have been possible without their help and guidance.

I would also like to thank my committee members, Pr. Rob Hierons, Pr. Stephan Merz, Dr. Ana Cavalcanti, Pr. Pascale Le Gall and Pr. Claude Marché for serving as my committee members. I also want to thank you for letting my defense be an enjoyable moment, and for all your brilliant comments and suggestions. Many thanks go in particular to Dr. Ana Cavalcanti and the colleagues from the university of York for their help, comments and suggestions during my PhD and also during my visits to York.

The past and present members of the Fortesse group have contributed immensely to my personal and professional development. I have had the pleasure to work with or alongside them. The group has been a source of friendships as well as good advice and collaboration.

In addition, I have been very privileged to get to know and to collaborate with many other great people who were or became friends over the last several years. I learned a lot from them about life, research, how to tackle new problems and how to develop techniques to solve them. So to all my friends and colleagues: thank you.

After and above all, I would like to thank my parents for instilling in me the love of science and also for their continuous support and patience at all times. I would like also to thank my sister and brothers. They were always supporting me and encouraging me with their best wishes.

# Contents

# List of Figures



# List of Tables

# Resumé

Le travail présenté dans cette thèse est une contribution aux méthodes formelles de spécification et de vérification. Les spécifications formelles sont utilisées pour décrire un logiciel, ou plus généralement un système, d'une manière mathématique sans ambiguïté. Des techniques de vérification formelle sont définies sur la base de ces spécifications afin d'assurer l'exactitude d'un système donné. Cependant, les méthodes formelles ne sont souvent pas pratique et facile à utiliser dans des systèmes réels. L'une des raisons est que de nombreux formalismes de spécification ne sont pas assez riches pour couvrir à la fois les exigences orientées données et orientées comportement. Certains langages de spécification ont été proposés pour couvrir ce genre d'exigences. Le langage *Circus* se distingue parmi ces langues par une syntaxe et une sémantique riche et complètement intégrées.

L'objectif de cette thèse est de fournir un cadre formel pour la spécification et la vérification de systèmes complexes. Les spécifications sont écrites en *Circus* et la vérification est effectuée soit par des tests ou par des preuves de théorèmes. Des environnement similaires de spécification et de vérification ont déjà été proposées dans la littérature. Une spécificité de notre approche est de combiner des preuves de théorème avec la génération de test. En outre, la plupart des méthodes de génération de tests sont basés sur une caractérisation syntaxique des langages étudiés. Notre environnement est différent car il est basé sur la sémantique dénotationnelle et opérationnelle de *Circus*. L'assistant de preuves Isabelle/HOL constitue la plate-forme formelle au dessus de laquelle nous avons construit notre environnement de spécification et de vérification.

La première contribution principale de notre travail est l'environnement formel de spécification et de preuve Isabelle/*Circus*, basé sur la sémantique dénotationnelle de *Circus*. Sur la base de Isabelle/HOL nous avons fourni une intégration vérifiée de UTP, la base de la sémantique de *Circus*. Cette intégration est utilisée pour formaliser la sémantique dénotationnelle du langage *Circus*. L'environnement Isabelle/*Circus* associe à cette sémantique des outils de *parsing* qui aident à écrire des spécifications *Circus*. Le support de preuve de Isabelle/HOL peut être utilisée directement pour raisonner sur ces spécifications grâce à la représentation *superficielle* de la sémantique (shallow embedding). Nous présentons une application de l'environnement à des preuves de raffinement sur des processus *Circus* (impliquant à

la fois des données et des aspects comportementaux). Une version de Isabelle/*Circus* est déjà publiée dans le cadre de l'AFP (archives de preuves formelles) d'Isabelle [1].

La deuxième contribution est l'environnement de test *CirTA* construit au dessus de Isabelle/*Circus*. Cet environnement fournit deux tactiques de génération de tests symboliques qui permettent la vérification de deux notions de raffinement: l'inclusion des traces et la réduction de blocages. L'environnement est basé sur une formalisation symbolique de la sémantique opérationnelle de *Circus* avec Isabelle/*Circus*. Plusieurs définitions symboliques et tactiques de génération de test sont définis dans le cadre de *CirTA*. L'infrastructure formelle permet de représenter explicitement les théories de test ainsi que les hypothèses de sélection de test. Des techniques de preuve et de calculs symboliques sont la base des tactiques de génération de test.

L'environnement de génération de test a été utilisé dans une étude de cas pour tester un système existant de contrôle de message. Une spécification du système est écrite en *Circus*, et est utilisé pour générer des tests pour les deux relations de conformité définies pour *Circus*. Les tests sont ensuite compilées sous forme de méthodes de test JUnit qui sont ensuite exécutées sur une implémentation Java du système étudié.

Cette thèse est une importante étape vers, d'une part, le développement d'outils de tests sophistiqués utilisant des preuves de propriétés de la spécification pour réaliser et justifier des stratégies de test et, d'autre part, l'intégration de test et de preuves dans des développements logiciels formellement vérifiées.

## Mots clefs

Spécification et vérification de logiciels, preuves de théorèmes, test formel.

[1] http://afp.sourceforge.net/entries/Circus.shtml

# Abstract

The work presented in this thesis is a contribution to formal specification and verification methods. Formal specifications are used to describe a software, or more generally a system, in a mathematical unambiguous way. Formal verification techniques are defined on the basis of these specifications to ensure the correctness of the resulting system. However, formal methods are often not convenient and easy to use in real system developments. One of the reasons is that many specification formalisms are not rich enough to cover both data-oriented and behavioral requirements. Some specification languages were proposed to cover this kind of requirements. The *Circus* language distinguishes itself among these languages by a rich syntax and a fully integrated semantics.

The aim of this thesis is to provide a formal environment for specifying and verifying complex systems. Specifications are written in *Circus* and verification is performed either by testing or by theorem proving. Similar specifications and verification environment have already been proposed. A specificity of our approach is to combine support for proofs and test generation. Moreover, most test-generation methods are based on a syntactic characterization of the studied languages. Our proposed environment is different since it is based on the denotational and operational semantics of *Circus*. The Isabelle/HOL theorem prover is the formal platform on top of which we build our specification and verification environment.

The first main contribution of our work is the Isabelle/*Circus* specification and proof environment based on the denotational semantics of *Circus*. On top of Isabelle/HOL we provide a machine-checked shallow embedding of the UTP, the semantic basis of *Circus*. This embedding is used to formalize the denotational semantics of the *Circus* language. The Isabelle/*Circus* environment associates to this semantics some parsing facilities that help writing *Circus* specifications. The proof support of Isabelle/HOL can be used directly to reason about these specifications thanks to the shallow embedding of the semantics. We present an application of the environment to refinement proofs about *Circus* processes (involving both data and behavioral aspects). A version of the Isabelle/*Circus* is already released as a part of the AFP (archive of formal proofs) of Isabelle [2].

[2] http://afp.sourceforge.net/entries/Circus.shtml

The second main contribution is the *CirTA* testing framework built on top of Isabelle/*Circus*. The framework provides two symbolic test-generation tactics that allow checking two notions of refinement: traces inclusion and deadlocks reduction. The framework is based on a shallow symbolic formalization of the operational semantics of *Circus* using Isabelle/*Circus*. Several symbolic definitions and test-generation tactics are defined in the *CirTA* framework. The formal infrastructure allows us to represent explicitly test theories as well as test selection hypothesis. Proof techniques and symbolic computations are the basis of the test-generation tactics.

The test-generation environment was used for a case study to test an existing message monitoring system. A specification of the system is written in *Circus*, and used to generate tests following the defined conformance relations. The tests are then compiled in the form of JUnit test methods and executed against a Java implementation of the monitoring system.

This thesis is a step towards, on one hand, the development of sophisticated testing tools making use of proof techniques and, on the other hand, the integration of testing and proof within formally verified software developments.

## Keywords

Software specification and verification, theorem proving, formal testing.

CHAPTER 1

# Introduction

Software and hardware systems continue increasing in size and complexity. Accordingly, the number of bugs is also increasing in these systems. The testing activity aims at finding these bugs. Formal testing denotes the use of formal methods and techniques in testing. Formal methods include mathematically based languages, techniques and tools for specifying and verifying software and hardware systems.

In formal testing, different formalisms can be used to describe formally the desired ("correct") behavior of the system. Specification languages are widely used as description formalisms in specification-based testing techniques. In order to cover the different aspects of the system behavior, different specification languages can be used. Some data-oriented languages focus essentially on complex data representation (*e.g.* Z). Other languages deal more with complex behavioral aspects of the system (*e.g.* CCS and CSP). Since real systems combine complex data and behavior, some languages have been proposed to cover these two aspects.

It is the case of the *Circus* language, which combines Z-like data representations with a CSP-like process algebra. It involves also a particular notion of refinement. The language comes with well established denotational and operational semantics. A formal testing theory is defined for *Circus* on the basis of its semantics.

A natural way of implementing formal methods and tools is by using theorem provers. These provers offer a formal "trusted" basis for any other formal development. One of the most known theorem provers is the Isabelle/HOL proof assistant for higher order logics.

## 1.1 Motivations

Dijkstra stated that "testing can be used to show the presence of bugs, but never to show their absence". One objective of formal testing is to show that this thesis is not always true. In fact, if testing covers all possible behaviors of the system, and under some hypotheses, it can ensure the absence of bugs. This testing approach is also called exhaustive testing. It combines testing and proof techniques to define a notion of correctness *i.e.* the absence of bugs.

A system is said to be correct *w.r.t.* a specification if it behaves satisfactorily *w.r.t.* this specification. We say then that the system is *conform* to the specification for a given *conformance* relation. A test theorem states that, for a pair of test cases and test hypotheses, the system behaves correctly *w.r.t.* a given specification.

The use of theorem provers makes possible the statement of test theorems and hypotheses, and provides a strong basis for testing strategies. Theorem provers offer also a powerful logical formal basis. This can be used to formalize rich specification languages such as *Circus*.

In this thesis we explore the combination of these two ideas *i.e.* the use of a theorem prover as a basis for specifying and testing complex systems. The *Circus* language, based on its semantics, is encoded in the Isabelle/HOL theorem prover. The prover is used also as a basis for a test generation system for *Circus*.

## 1.2 Proposal and Contributions

The goal of this work is to provide a formal and comprehensive framework for specifying, testing and reasoning on complex systems. We choose the *Circus* specification language to cover the different aspects of these systems. We believe that *Circus* provides an exemplary combination of both complex data and behavior descriptions. Our testing approach is based on the theorem prover Isabelle/HOL. It is used to formalize the semantics of *Circus* and its testing theory. It is also used to generate exhaustive tests from *Circus* specifications.

In this work, we aimed at a semantics-based approach for specifying, verifying and testing. Traditionally, specification languages are defined formally with well established semantics and properties. Some tools can be developed for these languages, covering parsing, type-checking, animations or test generation. The problem is that, in most cases, there is a gap between formal "pen and paper" definitions and the, usually, unverified tools. Moreover, the tools are usually syntax-based, with little or implicit semantic support. In our work, the gap between the theory and the tools is extremely reduced. By using the Isabelle/HOL formal environment, all formal definitions can be mechanized and implemented. This formal environment makes it also possible to associate semantics to the syntax. Thus, it is possible to perform more elaborated reasoning on the specifications.

This thesis presents a pioneering work in the field of formal specification, verification and testing environments. The formal semantics-based representation of a complex and rich specification language in a theorem prover is an important achievement. The main contributions can be summarized in five parts:

- A first contribution of this work is the shallow embedding of UTP, the semantics basis of *Circus*, in Isabelle/HOL.

- The representation of the denotational semantics of *Circus* on top of UTP is also an important contribution of this work, with the definition of several key notions like state variables, scoping, communication channels and name sets.

- The verification environment Isabelle/*Circus*, based on this representation, is a proof environment for *Circus* that can be used to reason about *Circus* specifications. An interesting exercise is explored, where the environment is expanded with some refinement rules over *Circus* processes.

- Another substantial contribution of this work is the representation of the operational semantics and the testing theory of *Circus*. The operational semantics rules are defined for *Circus* actions and some useful inverted rules are derived from them. The testing theory contains the definitions of many useful notions (*e.g.* traces and initials) as well as some testing theorems. These definitions are based on a shallow symbolic representations and manipulation based on Isabelle's symbolic facilities. Automatic test generation tactics are defined for *Circus* processes.

- The last but not least contribution is the application of the test environment to a realistic case study. The target application is a message monitoring module of a medical remote monitoring system. A specification of the system is written in *Circus*, then used as a basis for test generation. The generated tests are executed, using JUnit on the Java implementation of the system.

## 1.3 Outline

The aim of this thesis is to introduce a semantics-based testing approach and a formal testing environment for *Circus*. This is realized in two steps, covered by two main contributions. The first step is the definition of a formal specification and verification environment for *Circus*. The second step, presents a testing framework for *Circus* on top of the specification environment. The thesis is organized around these two main contributions. It is composed of five chapters (2-6) organized as follows:

**Chapter 2: Formal methods and testing**

In this first chapter we introduce some aspects of complementarity between formal methods and testing, with a particular regard for formal testing. After some background on testing theories, we discuss the state of the art and present the major formal testing techniques. The relation between traditional verification techniques and testing is discussed as a special kind of complementarity. In particular, we study some model checkers and theorem provers used for test generation.

**Chapter 3: Context**

We introduce in this chapter the two main concepts on which the thesis is built. The first concept is the formal specification language *Circus* and the second is the Isabelle/HOL proof assistant. In the first part of this chapter, we introduce the syntax and the semantics of *Circus*. These semantics are based on a semantic framework called UTP (Unifying Theories of Programming) which is also introduced. The first part of the chapter is devoted to the Isabelle generic theorem prover. We focus on the extension of the prover with higher order logics called Isabelle/HOL. We introduce some extensions of Isabelle/HOL which are related to our work: HOL-Z, HOL-CSP and HOL-TestGen. Finally, the chapter summarizes some general considerations that guide the following chapters.

**Chapter 4: Isabelle/Circus**

This chapter introduces the first main contribution of the thesis: the Isabelle/*Circus* specification and verification environment. In the first part of this chapter, we discuss some conceptual choices of representation. Then, we develop our embedding of a substantial part of UTP constructs on top of Isabelle/HOL. The definition of the *Circus* denotational semantics is introduced on the basis of the UTP embedding. The whole semantics enriched with a high level interface forms our specification environment Isabelle/*Circus*. This environment allows for reasoning on *Circus* specifications using the prover facilities.

We present a first application of the Isabelle/*Circus* environment to refinement proofs. We introduce the notion of data and action refinement in *Circus* as well as some general refinement laws. An example of refinement proof is illustrated on a small example.

**Chapter 5: Semantics based testing**

The second main contribution of the thesis is explained in this chapter. Based on the Isabelle/*Circus* environment, the *CirTA* testing framework is defined for *Circus* specifications. We introduce first our approach of theorem prover-based test generation

inspired from HOL-TestGen. An important part of this chapter is then devoted to the operational semantics of *Circus*. Our representation of the symbolic definitions and manipulations is also discussed. The symbolic testing approach is explained and the main definitions are introduced. A small example is used to illustrate the test generation tactics presented in this chapter.

**Chapter 6: Case study**

The last chapter illustrates with a case study the application of the proposed framework. A *Circus* specification of an existing message monitoring system is provided and used for test generation. JUnit test procedures are automatically extracted from the concrete test cases generated using *CirTA*. These test procedures are executed against the existing Java implementation of the monitoring system.

CHAPTER 2

# Formal Methods and Testing

## Contents



## 2.1 Introduction

Formal methods and software testing are not very old friends: they have even been seen as rivals. For several years, they had failed to converge. Between the late seventies and early eighties, some propositions for using both approaches together arose. Since then, formal methods and software testing are increasingly integrated in what is now known as *formal testing*.

Formal methods can be defined as mathematically based languages, techniques and tools for specifying and verifying software and hardware systems. This encloses mainly two kinds of formal methods: formal specification languages and formal verification techniques. We believe that formal testing is one of the formal verification

techniques. However, we will use the word verification in this chapter to talk about traditional formal verification techniques and especially model checking and theorem proving.

Formal testing gather all the impacts and contributions that formal methods can offer to testing. These contributions may be distinguished in two categories according to their original purpose. First, the formal definitions and techniques introduced in the context of testing, including test theories, test generation, test oracles and verdicts definitions. The second category includes formal methods and techniques developed for other purposes but reused for testing. This includes essentially formal specification languages used as basis for testing and formal verification techniques reused for testing.

This chapter gives an overview of these different contributions. Section 2.2 introduces formal testing definitions and techniques based on some formal specification languages. Section 2.3 exposes some ways of reusing verification techniques in the context of testing.

## 2.2 Formal Testing

Recently, formal methods have been widely used in testing; several ways of integration have been explored. When addressing the testing problem in a formal context, one need to define a formal testing framework. This framework is defined over three stages: test definition, generation and evaluation.

1. First, a test definition should be introduced according to what needs to be tested, this is usually done by introducing testing theories. This stage is explained in section 2.2.1.

2. Then, according to the test definition, tests are generated and executed against the tested system. Different test generation techniques are introduced in section 2.2.2.

3. Finally, the test results have to be evaluated in order to decide if the system have passed a test or not. This is done by defining test oracles and verdicts illustrated in section 2.2.3.

Formal testing techniques are usually based on a formal model or specifications. These techniques are called model-based testing or specification-based testing. In the context of this chapter, model-based testing and specification-based testing will be used interchangeably to address testing techniques based on a formal description.

### 2.2.1 Background on testing theories

In formal testing, the formal definition of a testing framework is fundamental. A meta-description of the framework is sketched in Figure 2.1.

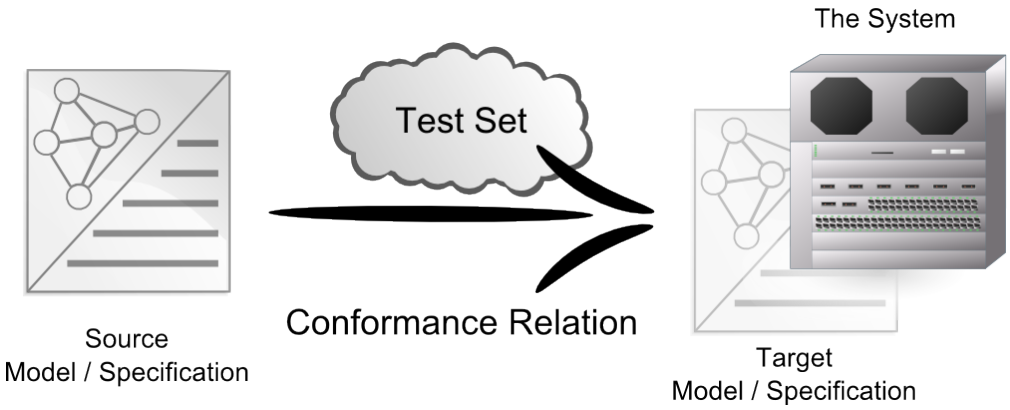



Figure 2.1: Testing framework

Such a framework is based on the relation a given system and specification must satisfy. This *satisfaction* relation depends on the objective of the testing framework. For a given relation and a specification formalism some *exhaustive* test set can be defined. Informally, exhaustive means that this test set covers all the possibilities described in the specification. For example, when using a behavioral language, an exhaustive test set covers all the possible behaviors of the specification. The notion of exhaustivity is stated more precisely in the sequel.

However, since systems and specifications are of very different nature [Gau11], the satisfaction relation is formalized by a conformance relation defined between two models (specifications) often described in the same formalism, complemented by some hypotheses on the system. This is very important in realistic testing scenarios, since models are not real systems (*the map is not the territory*). As the entity under test is a real system, one or more assumptions must be made to ensure that this system can be considered/observed as a model (specification) in order to be comparable with the original model (specification). These assumptions are commonly known as *testing (testability) hypotheses* and are part of the testing framework. In the rest of this chapter, the source model will refer to the original defined model and the target model to the assumed – but unknown – model (see Figure 2.1).

One of the first testing theories was defined in the work of de Nicola and Hennessy [dNH83]. The authors investigated the use of testing to check equivalence between formally specified processes. They defined three equivalence and preorder relations based on two verdicts *successful* and *unsuccessful*. The authors applied their approach to the specification language CCS. This work covers preorder rela-

tions between models. It does not address real system testing: for that, adequate testability hypotheses should be introduced.

Before the testing theory de Nicola and Hennessy, one of the early testing frameworks was underlying the W-method proposed by Chow in the late seventies [Cho78]. In this work, the author considered finite-state machines (FSM) as source models. The tested system is observable as if it was equivalent to an unknown FSM model. This unknown model is used as target model for the conformance relation in the testing theory. Several other works tackle the same problem using different source and target models. To consider the system under test (SUT) as a model, these methods make, explicitly or not, some assumptions on the SUT. For instance, in [Cho78, LY94, LY96] one important assumption is that the SUT behaves like an unknown FSM with a given number of states.

More recent works, notably [BGM91, Gau95], introduced explicitly these assumptions under the name of *testability hypotheses*. This notion is fundamental when testing systems using formal specifications. If the SUT is not testable, the testing activity is useless because its results make no sense. An example of non testable system would be a demonic one that behaves well *w.r.t.* the formal specification during testing and, maliciously or not, changes its behavior afterwards.

In [Hie02, Hie09], the author rightly pointed out that the notion of testability hypotheses is very similar to the notion of *fault domains*. A fault domain is a set of models such that the SUT is believed to behave like an unknown model of this set. Therefore, the fault domain can be defined as the set of all the models that satisfy the testability hypotheses.

Testability hypotheses, associated to a good formalization of the exhaustive test set, can be used to guarantee correctness (conformance). This somewhat goes against the famous aphorism of Dijkstra: "testing can be used to show the presence of bugs, but never to show their absence". Formal (*exhaustive*) testing can be used to show correctness *i.e.* the absence of bugs: this requires the testability hypotheses to be satisfied. Globally, one can state a correctness formula as follows:

> For a system under test SUT and an exhaustive test set T defined from a specification S and a given conformance relation, we have:
>
> *If the testability hypotheses hold then SUT is conform to S if and only if SUT passes all tests in T.*

Since the exhaustive test set is generally infinite, other *selection hypotheses* need to be defined in addition to testability hypotheses. These selection hypotheses allows selections of the tests to be executed from the exhaustive test set. The weakest selection hypothesis one can imagine is that the SUT only satisfies the testability hypotheses. This means that **all tests** in the exhaustive test set should be executed.

The strongest hypothesis assumes that the SUT behaves correctly *w.r.t.* the specification, this means that **no test** needs to be executed. Proving this hypothesis is equivalent to a correctness proof on the SUT.

In practice, these two cases are not directly usable: if the exhaustive test set is infinite, which is the usual case, the selection of all tests is not feasible; if the correctness proof is impossible: for instance in a black-box testing context where the details of the SUT are hidden, tests are necessary. Between these two extreme hypotheses, quite a number of selection hypotheses can be defined to select more reasonable number of tests.

Some examples of selection hypotheses were studied in the literature:

- One can make the hypothesis that the SUT will behave uniformly for a given set of data values. This is called *uniformity hypothesis* [BGM91, Gau95]: it allows the selection of a subset of data values that will represent the whole set of values.
- Another hypothesis can be stated if the behavior of the SUT shows some regularity *w.r.t.* some properties. This *regularity hypothesis* [BGM91, Gau95] allows the selection of a subset of tests up to a given regularity limit to represent the whole set.
- In [Pha94] some independence and fairness hypotheses were presented for the test of communication protocols against Input-Output State Machines specifications.

The notion of selection hypotheses is very close to the notion of *test (coverage) criteria*. A test criterion is a property that states if the test is sufficient or not. The relation between test hypotheses and criteria is explained in [Hie02].

Gaudel [Gau10] presented a generic definition of this testing framework. This definition was instantiated for different specification formalisms *e.g.* algebraic specifications, object-oriented Petri nets [PBB98], LUSTRE [MA00], LOTOS [GJ98] and FSM and LTS [LG02]. We sketch a formalization of this generic testing framework in Isabelle/HOL as follows:

```
1 type_synonym 'a test = "'a"
2 type_synonym 'a test_set = "'a test set"
```

The test type is defined above as a generic type that can be instantiated for concrete situations. The test set type is defined as a set of element of the generic type test. For an application to FSM testing for instance, the generic test type can be instantiated to a list of input/output symbols.

```
type_synonym 'b SUT = 'b
type_synonym 'b testability_hypothesis = "'b SUT ⇒bool"
```

The type of the systems under test, SUT, is also defined as a generic type. Its instantiation depends on the nature of the concretely tested systems. The type of a testability hypotheses is defined as a predicate over the SUT.

```
type_synonym 'a selection_criterion = "'a test_set ⇒'a test_set"
type_synonym ('a, 'b) verdict = "'b SUT ⇒'a test ⇒bool"
```

Above, we state that a test selection criterion describes the selection of a subset of tests from a given test set and that the verdict is a predicate that checks if a SUT passes a given test.

```
definition
Test_verdict::"('a, 'b) verdict ⇒'b SUT ⇒'a test_set ⇒bool" where
"Test_verdict pass sut ts = (∀ t∈ts. pass sut t)"
```

The definition of the verdict is generalized to a predicate over a test set in the definition `Test_verdict`.

```
type_synonym 'c Spec = 'c
type_synonym ('c, 'a) SBTS = "'c Spec ⇒'a test_set"
```

As for SUT, the type of a specification is generic and will be instantiated depending on the specification formalism. A specification-based test set type (SBTS) is defined as a function type that returns a test set from a given specification.

Now we consider selection hypotheses:

```
definition
selection_hypothesis::"'a selection_criterion ⇒'b SUT ⇒
                       'a test_set ⇒('a, 'b) verdict ⇒bool"
where
"selection_hypothesis sc sut ts pass =
            (∀ t∈(sc ts). pass sut t) ⟶ Test_verdict pass sut ts"
```

A selection hypothesis is defined from a test selection criterion, a SUT, a test set and a verdict function. It states that the SUT passes the test set if it passes all the tests selected by the selection criterion.

Finally, the conformance relation type is defined as a relation over a specification and a SUT. Exhaustivity of a specification-based test set is then defined *w.r.t.* a conformance relation. Given a conformance relation, the pair (test set, hypothesis) is exhaustive *w.r.t.* this relation if and only if whenever the testability hypothesis holds for a given SUT, this SUT conforms to a specification if it passes the test set.

```
type_synonym
('b, 'c) conformance_relation = "'b SUT ⇒'c Spec ⇒bool"

definition
Exhaustive::"('c, 'a) SBTS ⇒'b testability_hypothesis ⇒
               ('a, 'b) verdict ⇒('b, 'c) conformance_relation
                 'b SUT ⇒'c Spec ⇒bool"
where
"Exhaustive ts Hmin pass conf_rel sut spec =
    (conf_rel sut spec ≡ Hmin sut ∧ Test_verdict pass sut (ts spec))"
```

where `Hmin` is the minimal testability hypothesis stated on the SUT.

This framework must be instantiated for specific specification formalisms, classes of systems under test and the corresponding testability hypotheses, and the conformance relation.

### 2.2.2 Test generation from formal specifications

One of the most significant contribution of formal methods to testing is the use of formal specification languages. In the early stages of system development, abstract specifications of the desired system are performed. These specifications are written in a formal specification language or formalism. The advantage of such specification languages is their associated well defined syntax and semantics. These languages are also usually supported with tools like editors, parsers, type checkers and animators.

Formal specification languages were used in several testing techniques, based on different test theories and hypotheses. In the following, we discuss some of these techniques, classified according to the used specification language. More exhaustive and detailed studies and surveys can be found for instance in [CS94, LY96, Pet01, HBB+09, Gau10]. According to the main covered aspect (data or behavior), we classify specification languages in three classes: (i) data-oriented languages, (ii) behavioral languages and (iii) combined languages.

#### 2.2.2.1 Data-oriented languages

The first class of specification languages is the data-oriented languages also known as model-based languages. Their main concern is to represent complex data structures and operations on them. The system is specified by representing its state, and operations that update it. We present some of these languages in the sequel (VDM, Z, B and algebraic specifications) as well as some test generation techniques based on them.

### VDM

*Vienna Development Method* specification language (VDM-SL) [Jon86, VDM] is a model-based specification language. Specifications define an abstract state and operations which are relations on this state defined in terms of pre/post conditions. Operations can be defined with inputs and can return outputs in addition to state variables. The preconditions of operations are first order predicates over the operations inputs. The postconditions give a relation between the operations inputs, outputs, the old and the new state.

The most significant formal testing approach for VDM was proposed by Dick and Faivre [DF93]. Even if this work uses VDM as source specification language, it can be generalized and applied for all model-based specification languages (Z for instance). As explained above, VDM operations are expressed in term of first-order predicates describing relations between system states. The test selection is performed by a partitioning strategy, where the input space is partitioned into sub-domains, giving one test per sub-domain. This is done by reducing the pre and post conditions into their disjunctive normal form (DNF), describing the set of disjoint sub-domains. The result of the partition of all state expressions to DNF is used to extract a finite state automaton. This automaton is then used to generate test sequences with the aim of path coverage. This approach was implemented in a Prolog based tool [DF93] that also provided a VDM editor and type-checker. The kernel was the transformation of predicates into DNF. Test selection and automaton extraction were not automated. The main problems of this type of approaches is the explosion of the DNF size and the introduction of infeasible cases resulting from the DNF transformation.

### Z

The Z specification language [Spi88, Spi92, WD96] is based on *schema* definitions and operations to specify systems. The state (with complex data types) is represented by *state schemas*. Operations on the state are defined in terms of *operation schemas*. A schema consists of variable declarations associated to a predicate that constrains their values. The predicate of a state schema defines the state invariant. The predicate of an operation schema describes this operation in a pre-post condition style. Variable names represent their initial values while names decorated with $'$ represent their final values *i.e.* after the operation is performed. The first ISO standardization of the Z languages was published in 2002.

An example of a Z specification is the birthday book example presented in [Spi92] and shown below. First, abstract types *NAME* and *DATE* are declared. Three schema expressions (*BirthdayBook*, *InitBirthday* and *AddBirthday*) are then defined based on these abstract types.

$[NAME, DATE]$

*BirthdayBook*

$$known : \mathbb{P}\, NAME$$
$$birthday : NAME \pfun DATE$$
$$known = \mathrm{dom}\, birthday$$

*InitBirthday*

$$BirthdayBook$$
$$known = \emptyset$$

*AddBirthday*

$$\Delta BirthdayBook$$
$$name? : NAME$$
$$date? : DATE$$
$$name? \notin known$$
$$birthday' = birthday \cup \{name? \mapsto date?\}$$

The first schema *BirthdayBook* defines the system state, using a set of known names and a partial function that returns a birthday for a given name. The associated predicate states that known names are those for which a birthday is defined. The second schema *InitBirthday* describes the initial state of the system in which the birthday book is empty. The third schema *AddBirthday* defines an operation over the state. It adds a name and its corresponding birthday to the state. The precondition of the operation is that the new name is not yet in the birthday book. The postcondition describes the resulting state by adding the new name and birthday to the initial state.

Most of the Z based test generation approaches are based on the state partitioning strategy described for VDM. Hall [Hal88] was the first to discuss a general approach to testing from Z specifications, aiming at test generation and test selection automation.

Next, Amla and Ammann [AA92] applied a category partitioning method to Z specifications somewhat similar to [DF93] for VDM. Then Ammann and Offutt [AO94] proposed a method to specify combinations of category partitions in form of coverage criterion.

Stocks and Carrington's *Test Template Framework (TTF)* [CS94, SC96] offers a well-known methodology for testing based on Z specifications. It uses a partition analysis strategy called *domain propagation* for state partitioning. For each operation, a set of possible input sub-domains is defined from the specification of the operation. In a complex expression, the sub-domains of each of its operation is propagated to build a set of all possible sub-domains. Recently, this method was (partially) implemented by [CM09] in a model-based testing tool. This tool relies on the CZT framework for parsing, type-checking and animating Z specifications. The main problem of this tool is the lack of efficient constraint solving.

Helke et al. [HNS97] provided an embedding of Z in Isabelle/HOL theorem prover. This embedding is used as a basis for test generation from Z specifications using the partitioning strategy. The DNF computation and unsatisfiable and redundant cases elimination were automated using the theorem prover.

Hierons [Hie97] defined a decomposition strategy by rewriting Z specifications to a flattened expressions based on first order predicate logic and set theory. The resulting expressions are transformed to a disjunction of pre/post-condition expressions that will be used for test generation. This strategy reduces the number of the generated sub-domains comparing to a full DNF decomposition.

Singh et al. [SCS97] proposed a combination of the classification-tree method with the DNF approach. A classification-tree that contains test cases is computed from a high-level description of the system. A DNF decomposition is then applied to these test cases to obtain more refined test cases.

### B

The B-method is a software development method focused on the refinement of specifications written in the B notation [Abr96, Sch01]. B specifications can be described in terms of abstract machines or as refinements of other B specifications. An abstract machine defines the state variables, state invariants, initialization and operations over the state.

Aertryck et al. [vABM97] investigated test (suite) generation from formal (B) specifications. The proposed approach is based on some predefined or user-defined test generation strategies described in a so called test case specifications *tcs*. Constraint solving is used to instantiate test data that satisfy each test case specification. This approach was implemented in a tool called CASTING.

Legeard et al. [LPU02] proposed a boundary testing method[1] for B and Z specifications. Boundary goals are computed from the DNF of the model, then boundary states are computed following these goals. For each boundary state, all possible behaviors (starting from it) are tested. The BZ-TT environment provides an im-

[1]Boundary testing is a testing activity that targets the boundaries or limit values of the test input domain.

plementation of this testing method including powerful specific constraint solving techniques.

### Algebraic specifications

Systems can be described in terms of their algebraic properties, using the so-called algebraic specifications. Algebraic specifications are defined by sorts, operations and some properties (expressed as axioms). Sorts are the data types specified in the given algebraic specification. The operations (functions) signatures describe the syntax of the specification. The semantics of the specification is given in terms of axioms on the operations describing their properties. Examples of algebraic specification languages are CASL [ABK+02], ACT-ONE [EFH83] and OBJ [GT79].

As said above, algebraic specifications allows the description of abstract data types as constants, operations and then properties (axioms). Testing techniques based on algebraic specifications aim at verifying that an implementation of a specified data type satisfies its axioms. One of the first works on testing using algebraic specifications was done by Gannon et al. [GMH81]. The authors developed the DAISTS system, that allows for specifying, implementing and testing abstract data types. For a user-supplied inputs, the systems generates the corresponding outputs from the implementation of the data type operations. The specified axioms are then used directly as test oracles to determine if the outputs are correct. A notion of structural coverage criteria was also considered *e.g.* axioms and operations coverage.

A theory for testing from algebraic specifications was developed by Gaudel et al. [BCFG86, BGM91, DGM93, GLG08]. Testability hypotheses were expressed as well as several exhaustive test set definitions. These sets are based on the set of all the ground instantiations of the axioms in the specification. Due to the infinite nature of test sets, selection hypotheses (*e.g.* uniformity and regularity) were proposed. This approach was implemented in the LOFT tool and applied to an automatic subway software. This tool is based on constraint solving and predefined selection hypotheses.

Machado [Mac00] tackled the oracle problem in the case of algebraic specifications testing. Using the axioms of the specification as oracle reduces the problem to an equality problem, but can lead to comparing two values of a non observable sort. This oracle problem is explained with more details in Section 2.2.3.

### Logical specifications

Logics are widely used to describe the semantics of model-based specification languages. They are also used as an independent basic specification formalism to support logical reasoning on specifications. Theorem provers use logical basis to offer different specification and symbolic reasoning facilities.

Some testing techniques and tools were proposed for logical specifications, for example the HOL-TestGen [BW12] system based on the Isabelle/HOL theorem prover. This system is based on specifications written in Higher-Order Logic to generate test cases and test data. Some test selection hypotheses (uniformity and regularity) are stated explicitly in the system. This system is of particular importance to our work. It will be introduced with more details in 3.3.3.

#### 2.2.2.2 Behavioral languages

The second class of specification languages contains behavioral languages. These languages cover sophisticated behavioral aspects of reactive systems such as non-determinism and concurrency. In the following, we focus on two different kinds of behavioral languages: process algebras (CSP) and graph based formalisms (finite state machines and transition systems).

##### CSP

The Communicating Sequential Processes (CSP) [Hoa85, RHB97, Sch99b] is a well known process algebra. Reactive systems can be described in CSP in terms of communicating processes. Processes communicate by events performed by synchronization or value exchange over channels. CSP define a set of operators to describe internal and external choices, sequential and parallel compositions, conditionals and hiding. Communications are defined using a prefixing operator.

Testing based on CSP specifications was first studied by Peleska and Siegel [Pel96, Sie97]. The authors defined a testing method inspired by de Nicola and Hennessy's testing theory [dNH83]. They defined different refinement relations in terms of traces, refusals and divergences. Notions of *may* and *must* testing were introduced producing two verdicts *pass* and *fail*. The testing approach was implemented in the VVT-RT system, using the FDR [For05] model checker to retrieve the LTS representation of a CSP process.

Schneider [Sch99a, Sch99b] defined two conformance relations based on testing for CSP specifications. The author distinguished two categories of events: refusable and non-refusable, and introduced on this basis an operational characterization of process abstraction in term of testing.

Cavalcanti and Gaudel [CG07] introduced a testing theory for CSP based on the failures-divergences refinement. The proposed testing theory instantiates the generic testing theory defined in [Gau01] for CSP specifications. The authors characterized mandatory testability hypotheses for CSP-based testing. They also provided definitions for exhaustive test sets *w.r.t.* traces and failures refinement and considered some selection hypotheses.

Nogueira et al. [NSM08] introduced an other testing theory for CSP based on the testability hypotheses defined in [CG07] and a new conformance relation *cspio*.

This conformance relation is inspired from the *ioco* relation defined for Input-Output LTS. The authors proposed a test generation strategy based counter-examples generation for refinement model checking using FDR. Test purposes, described as CSP processes, were used for test selection.

### FSM

Finite state machines have been widely used as a basis model for several test generation techniques. In his seminal work, Moor [Moo56] introduced the problem of testing FSM using a concept used by physicists called *gedanken-experiments*. The global idea of an experiment is to interact with a system (sequential machine) in order to identify a transition diagram that describe its behavior.

Hennie [Hen64] introduced the concept of transition checking using checking sequences. His work was focused on fault detection experiments for sequential circuits represented by FSM. A checking sequence is an input sequence that starts from the initial state and determines if the target FSM is faulty. The studied FSM are assumed to have a distinguishing sequence, which is not always the case. A distinguishing sequence is a sequence of input for which the FSM returns a different output sequence for any different initial state. This distinguishing sequence is used to identify which state of the FSM corresponds to a given system state.

Inspired from the last work, number of FSM-based testing methods were proposed. Each method had special considerations and hypotheses on the tested system (*e.g.* the presence of an explicit reset or a distinguishing sequence). We already mentioned the W-method presented by Chow [Cho78] that used characterization sets as a basis for testing FSM. A characterization set (also called *W-set*) is the set of input sequences that can distinguish any two states of the FSM. This method uses a *testing tree* which is the result of flattening the FSM into a tree then computes minimal partial paths from this tree that cover the transitions. Test sequences are built by a concatenation of one of the computed paths with the elements of the characterization set.

Lee and Yannakakis addressed in their remarkable survey on FSM-based testing [LY94, LY96] the complexity of finding distinguishing sequences and Unique Input Output (UIO) sequences. The distinguishing sequence is used to identify the initial state of an FSM and the UIO sequence is used to verify if a given state is the initial state of an FSM. This notions are very useful when testing systems that behave like an FSM, particularly when the state of the system cannot be directly observed.

A classical testability hypothesis is considered when testing using FSM. It assumes the SUT to be equivalent to an unknown FSM which have the same input alphabet and the same number of states or a bigger but known number of states. Usually, the coverage criterion used in FSM-based testing is transition coverage since it subsumes the state coverage. Transition coverage is considered as a sufficient cri-

terion because each transition in a traditional FSM is independent from the other ones. The transition depends only from its source state and input symbol, covering a transition in one test is then sufficient.

Extended FSM were used to describe systems with state variables, operations over them and communicated values in inputs and outputs. Transitions may be guarded *i.e.* a condition over variable values is associated to each transition. EFSM can be classified as a combined specification formalism since they deal with both data and behavioral aspects.

Some test generation techniques transform the EFMS to an equivalent FSM by enumerating all possible data values. Due to the state explosion problem, symbolic evaluation techniques were applied in other EFSM-based testing techniques to deal with huge or infinite data types. These techniques depends usually on the symbolic traces of this EFSM to define symbolic tests. A symbolic trace is a sequence of symbolic transitions associated to a trace conditions (defined by the transitions guards). The problem with these symbolic techniques is the number of infeasible traces that can be generated. Some symbolic test generation methods use a constraint solver to eliminate infeasible traces early in the generation phase.

### LTS, IOTS/IOLTS

Labeled transition systems (LTS) are widely used as a semantic model for reactive systems. The graphical representation of LTS gives a better view of the different evolutions of the system. The state of the system is represented as nodes and the state transitions as labeled edges. Process algebra operational semantics is usually defined as an LTS that corresponds to the different transition rules. Transitions are labeled with actions (events) and states represent continuations after these actions. In classical LTS events represent undirected communications (synchronizations). In more elaborated models, communications are directed: one distinguishes input and output communications, for example IOTS (Input Output Transition Systems) [Tre96] and IOLTS (Input Output LTS) [JJ04].

Number of LTS-based testing techniques have been proposed (see [BT01] for an annotated bibliography). In general, when testing against LTS (or IOTS/IOLTS), two conformance relations can be used: *conf* and *ioco*. The *conf* relation [Bri88], whose source and target models are LTS, checks that all the deadlocks of the SUT are specified. The *ioco* relation [Tre96], whose source model is an IOTSL/IOLTS and target model is an input enabled IOTS/IOLTS, checks that the SUT never produces an unspecified output.

The formal testing framework for IOTS has been stated by Lestiennes and Gaudel [LG02], where an exhaustive test set definition for *ioco* was proposed. The authors also identified some testability hypotheses that must be considered when testing the *ioco* relation. First, the SUT is *input enabled i.e.* input actions are controlled by

the environment and cannot be refused by the system. Another hypothesis is the ability to observe *quiescence i.e.* when the SUT is waiting for an input from the environment and no output or internal actions can be performed.

Some tools have been developed implementing *ioco*-based test generation, we cite for instance TorX [TB03] and TGV [JJ04]. Other tools developed initially for process algebra based testing can also be used for LTS testing. This is a natural consequence of using LTS to represent the operational semantics of many process algebras (*e.g.* CSP and LOTOS).

Like for FSM, extended LTS were proposed to specify systems that deal with huge or infinite data types symbolically. STS (symbolic transition systems) [FTW06] also called SLTS (symbolic LTS) are transition systems extended with a notion of data and operation over it. In addition to states and labeled transitions, an STS contains some state and communication variables. A transition is labeled with a constraint, a communication and a set of operations over variables.

In [LG02] symbolic definitions of states, actions and trace were proposed for LTS extended with data types. Based on these definitions, a symbolic exhaustive test set was introduced for *ioco*. The presented testing theory contains also some selection hypotheses and a instantiation based on constraint solving techniques. A symbolic conformance relation called *sioco* was introduced in [FTW06] as a symbolic variant of *ioco* for STS. The authors proposed a symbolic state coverage criterion but symbolic test definitions or generation were not addressed yet.

#### 2.2.2.3 Combined languages

Despite the fact that data-oriented languages or behavioral languages were widely used to specify real systems, the latter usually involve both complex behaviors and complex data structures. Consequently, only one aspect of these systems can be specified at the same time. This is one of the reasons that justify the introduction of a class of combined languages. A combined language is, in most cases, a fusion of a data-oriented language and a process algebra. We introduce in the sequel some of these combined languages (LOTOS, SDL and Statecharts) and some testing techniques based on them. LOTOS, SDL and Statecharts are very similar to SLTS and EFSM. Consequently, several testing techniques for SLTS and EFSM were reused and adapted for testing based on these formalisms.

Test generation techniques based on combined languages are usually based on symbolic execution to deal with large and infinite data. This may introduce some unfeasible data values or behaviors. Efficient constraint solvers must be used in order to prune infeasible cases and also to instantiate concrete tests. Most of the test generation techniques for combined languages require powerful symbolic execution and constraint solving systems. This makes the tools implementing these techniques not very efficient and thus rarely used for real size applications.

### LOTOS

LOTOS (Language of Temporal Ordering Specification) [BB87, lot89] is a specification language combining algebraic specifications (based on ACT-ONE) an process algebras (based on CCS). Specifications in LOTOS are composed of two orthogonal parts for data and control. The data part contains abstract data types declaration in ACT-ONE. The control part is a description of the behavior in a process algebra called *basic* LOTOS, based on CCS. The so-called *full* LOTOS supports communications with the complex data types defined in the data part. The semantics of the two parts of the language are given separately. It is based on ACT-ONE semantics for the data part and an automata based operational semantics for the control part. Several extensions and variants of the language have been proposed, an example is LOTOS NT [Sig00].

Among the LOTOS based testing techniques, we cite for example Tripathy and Sarikaya's technique [TS91]. The authors proposed an approach to derive test cases from full LOTOS specifications for protocol conformance testing. LOTOS specifications were first translated into charts, a particular kind of transition systems. Tests are then derived from the resulting charts. Symbolic evaluation techniques were used to deal with infeasible test cases. This approach was implemented in the LOTEST test generation tool.

An other substantial work in this direction was developed in CADP [GLMS11], a toolbox for verifying asynchronous concurrent systems. It allows for efficient parsing, verifying and testing based on LOTOS specifications. Test generation is also based on the LTS semantics of the specifications.

Van der Schoot and Ural [vdSU95] introduced a data flow oriented test selection method for LOTOS. This method starts with static data flow analysis on the specification to identify input/output relations. Feasible tests are then generated to cover these relations using a set of inference rules.

### SDL

SDL (Specification and Description Language) [IT99] is a formal specification language for reactive and distributed systems. SDL is also object oriented and was originally proposed to describe telecommunication systems. Specifications presentation can either be textual or graphical in a finite state machine style. Abstract data type definitions are used to specify the static part of systems. A system is composed of blocks which are composed of communicating processes. Abstract State Machines were also used to define the semantics of SDL.

SDL was also used as basis for some test generation techniques especially in telecommunications area. In [GJK99] for example, an automated test generation technique for SDL specifications was proposed. It combines two existing tool supported test generation approaches. The first approach is based state space explo-

ration with heuristics and implemented in the TVEDA [GR98] tool. The second, based on an on-the-fly exploration of IOLTS models, is implemented in the TGV [JJ04] tool. The technique was implemented in a tool called *TestComposer*.

#### Statecharts

UML Statecharts [HG96] can be seen as object based finite state machines. They are very close to object based programming languages. Object oriented notions are used to describe data and the behavior is represented by a transition system. Statecharts are represented graphically as transition systems where (possibly guarded) transitions are labeled with pairs event/action. A semantics based on message passing Abstract State Machines was defined for Statecharts.

For Statecharts based testing, we refer for instance to the work by Hierons et al. [HSS01] on testing based on specifications that combine Statecharts and Z. The proposed approach produces an EFSM from the initial specification using *state abstraction*. The state abstraction applies Z state partitioning techniques to decompose operation domains. The resulting EFSM is used to generate test cases considering different test criteria.

Bogdanov and Holcombe [BH04] proposed a Statecharts based testing method supporting hierarchy and concurrency. They also proposed test selection criteria (refinement constraints) to reduce the size of test sets without weakening the test conclusions.

#### 2.2.2.4 Circus-based testing

In this thesis, the *Circus* language is used as a basis for formal specification and verification techniques. In order to support both static and dynamic aspects of systems, using a combined language is more convenient. Z provides a very good basis to describe complex system states and rich data structures. CSP offers well-established mechanisms to represent complex behavioral aspects like communications, sequencing and concurrency. *Circus* combines Z and CSP operators and also includes specification constructs and (Dijkstra like) guarded commands. It can be used at different levels of abstraction to write specifications, designs and programs.

Other sophisticated languages similar to *Circus* were already defined as combinations of a model based language and a process algebra. Examples of these combinations are CSP and Z (*e.g.* CSP-Z [Fis96], CSP-OZ [Fis97] and TCOZ [MD98]), CSP and B (*e.g.* CSP||B [ST05] and ProB [LB03]), CSP and CASL (*e.g.* CSP-CASL [Rog06]), CCS and Z (*e.g.* ZCCS [GS97] and CCZ [Gal96]), CCS and CASL (*e.g.* CCS-CASL [SAA02]) and CCS and ACT-ONE (*e.g.* LOTOS[BB87, lot89]).

Even if some of these languages are very close to *Circus*, the latter distinguishes itself for different reasons. *Circus* is provided with well defined denotational and operational semantics. Its denotational semantics is fully integrated in one unified

framework (UTP). The semantics of schema operations, CSP operators and commands are unified in this formalism. This makes it possible to reason about *Circus* as one entity without any need to separation. This integration allows also data and behavioral operations to be freely mixed in the specifications. For example, a schema operation can be prefixed by a communication then followed by a command. The operational semantics of *Circus* is also provided in terms of UTP. The UTP provides a well established relational framework for proving its soundness *w.r.t.* the denotational semantics.

Theories for refinement and testing were defined for *Circus* specifications. The refinement is based on the denotational semantics and cover both data and behavioral aspects. The symbolic testing theory is defined on the basis of the operational semantics of *Circus*.

In Section 3.2.3.2 the theory of testing based on *Circus* specifications is presented. Symbolic tests are defined *w.r.t.* two conformance relations inspired from process algebra testing definitions. These tests are based on a symbolic constrained version of events, traces, initials and acceptances. Symbolic variables are used to represent communicated values that can be of any (even infinite) type. Input and output domains are restricted by some constraints associated with each test.

Since the language and its tests definitions are based on both behavioral and data aspects, existing testing methods presented previously may be reused or adapted for *Circus*. The category partitioning and domain propagation techniques (defined for VDM and Z) can be applied on *Circus* symbolic tests. In fact, the input domain defined by the test constraint can be split to disjunct sub-domains by applying a DNF decomposition on its constraint. This will lead to possible selection hypotheses for these symbolic tests. Other selection hypotheses, that may provide more tests, can be defined using domain propagation techniques.

Some testing techniques based on process algebra (and especially CSP) inspired the definitions of tests for *Circus*. The particularity of *Circus* comparing to CSP is the manipulation of complex data types. This is covered in the definition of tests using symbolic execution techniques. A work that is of a particular similarity to the work presented in this thesis is the one by Helke et al. [HNS97] for CSP based testing. The similarity is that both works use Isabelle/HOL as a basis of the test generation framework. Infeasible and redundant test cases can be efficiently pruned using Isabelle and all constraint solvers it supports.

The *Circus* operational semantics associates to a *Circus* specification complex transition system with some similarities with EFSM and SLTS. In Section 3.2.3.2, the presented approach for symbolic test definition, selection and generation is somehow inspired from the works on testing based on EFSM and SLTS. However, these notions are not directly usable in *Circus* since its constructs are more rich and complex than EFSM and SLTS. In addition, the tested conformance relations addressed in both techniques are quite different.

### 2.2.3 Oracle and verdict

The *verdict* states if the tested system passes or not a given test. In many testing approaches three different verdicts are used *pass*, *fail* and *inconclusive*. The *pass* verdicts indicates that the SUT answered correctly to the test *i.e.* it conforms to the specification and the test objective is achieved. The *fail* verdict is emitted when the SUT fails in the test indicating a fault *w.r.t.* the specification. If the test objective is not achieved but no failure has be detected, the test verdict is *inconclusive*. When executing a particular test against the SUT, the latter may act in a (probably) correct way but different to what is defined in this test. This can happen for example in a nondeterministic situation if the SUT makes a different choice than what is proposed by the particular test.

The verdict is emitted by *oracles i.e.* automated or manual decision procedures that performs some interpretation of the test execution results. Defining such procedures is often very difficult; this is usually referred to as the *oracle problem*. This problem arises when there is an abstraction gap between the specification and the implementation: the interpretation of the results depends on some implementation choices unknown at the specification level. The oracle problem is usually reduced to checking some *equality* between the expected and the actual resulting value. This equality cannot always be directly expressed between values of an abstract data type and its representation in the SUT.

Equality over concrete data types (implementation) can be used as oracle. Unfortunately, this equality is usually a part of the implementation of the SUT and consequently is error-prone. In practice, only equality over some basic types (integers, booleans, ...) can be trusted since it is usually provided by the compiler. This can be expressed as a testability hypothesis called *oracle hypothesis*, stating that the equality over these basic data types is correctly implemented.

These basic data types are usually called *observable*; they are used to observe the implementation of the other *non-observable* data types. For this second kind of data types, one possible solution is to introduce observation functions. These functions compute an observational value from a non observable one, so the equality over the last can be used as oracle.

In the case of testing methods based on FSM, the verdict depends on the final state of the system after executing a test. The oracle problem is to determine without ambiguity this final state in order to emit the verdict; this is also known as state identification or observation. The only way to know the current state of a running SUT is by interacting with it and evaluating its outputs.

Some state observations techniques were defined for FSM; the most general one is the *separating family*. Each state of an FSM can be distinguished by observing the output sequences of a set of input sequences. The set of all these sequence set is called a *separating family* and it can always be defined for any FSM. A special case

of separating family is when the same input sequences set can be used to identify all states. In this case this set is called a *distinguishing set* and it can always be defined for any given FSM. If the distinguishing set contains only one sequence, the latter is called a *distinguishing sequence*. Unfortunately, deciding whether a given FSM has a distinguishing sequence is PSPACE-complete.

These techniques can be used as oracles to identify the final state after executing a test. Another related problem in testing based on FSM is to ensure that the SUT is in a given (initial) state before running a test. This problem is similar to the state identification problem, but it concerns more the testability hypotheses than the oracle. Unfortunately, state observation techniques cannot be used for this case, since they induce a state change. Some systems may have an explicit *reset* that can be used to put the system in its initial state. For FSM without a reliable reset, the so-called *homing sequences* are used to put the system in a known state. A homing sequence is an input sequence similar to the distinguishing sequence. The difference is that the homing sequence determines uniquely the arrival state of the system after executing it.

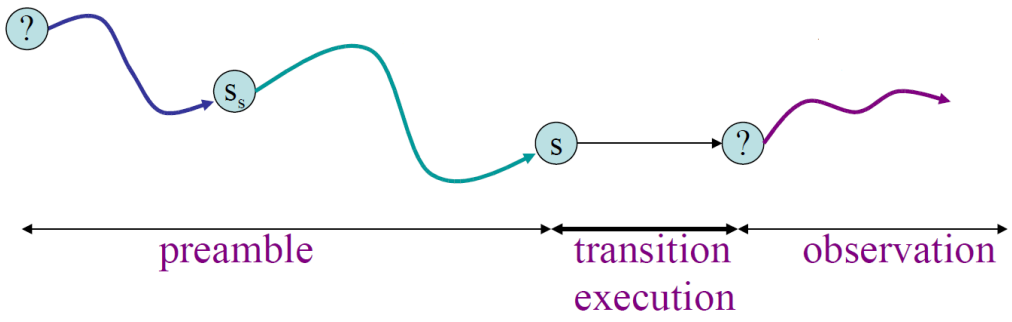



Figure 2.2: An FSM test example

An example of a complete test sequence for FSM based testing is given in figure 2.2. This sequence is composed of three parts: a preamble, a transition execution and then a state observation. Starting from an unknown state, a reset or a homing sequence is used to put the system in a known (initial) state. From this initial state, the system is executed against a given trace in order to reach the state to be tested. Once this state reached, the preamble part is achieved and the transition checking can be performed. This can be done by presenting an input to the system then observing its output and final state. This final state is unknown and can be observed by one of the state identification techniques introduced earlier (*i.e.* separating family, distinguishing set or distinguishing sequence). This observation part is an oracle for these kind of tests.

## 2.3 Verification and Testing

Even if they were assumed to be very different, verification and testing are becoming more and more closer in many aspects. Formal verification was considered as a static analysis with a full coverage of the model. Testing was seen as a dynamic activity that covers some aspects of the system. Recently, this distinction is becoming more and more fuzzy, with the introduction of dynamic verification techniques (*e.g.* software model checking) and the definition of formal exhaustive testing. Some works on the cross-fertilization of these two disciplines show that this distinction is actually unfair. In this section we present some of these "reconciling" works. Essentially, we give some applications of static program analysis, model checking and theorem proving techniques for testing.

Symbolic execution [Kin76] is one of the widely used static program analysis techniques. It provides a symbolic representation of a program, where variable values are replaced by symbols. Every possible program evolution is described by a path predicate expressed as a symbolic constraint. The set of all possible symbolic paths of a program can be represented in a symbolic tree. A symbolic path is said to be feasible if its constraint is satisfiable, otherwise it is infeasible. Symbolic execution is usually associated to constraint solving techniques, *e.g.* for checking feasibility or for symbolic values instantiations.

When testing against programs or specifications dealing with infinite objects or infinite type of objects, the number of possible test cases can be huge or infinite too. Symbolic representation and execution are used in order to represent these tests in a concise way. A symbolic test case can be represented for example as a symbolic execution path of a program. This unique test case, is a way to describe a huge number of cases where symbolic values are replaced by concrete ones. This kind of instantiation is performed with the help of constraint solving techniques.

Model checking is a static verification technique whose aim is to check that a model satisfies some properties. Models are finite state transition systems *e.g.* LTS or similar to them (Kripke structures, automata). Properties, sometimes called specifications, express some requirements that the model should preserve *e.g.* safety or liveness. Model checking is realized by an exhaustive exploration of the model with the goal of finding counter examples.

Exploiting the fact that model checkers find counter examples, some of them were used for test generation. Given a model and a property, model checking the negation of this property yields a counter example. This counter example represents a trace that satisfies the property and can be used to obtain test cases. Many model checkers have been used for test generation, as examples Java Pathfinder and Isabelle/HOL tools (Quickcheck and Nitpick). Java Pathfinder model checker has been used for generating test input from descriptions of method preconditions [VPK04]. The approach combines model checking, symbolic execution and con-

straint solving applied to Java programs. Isabelle/HOL theorem prover encloses tools for counter example generation used in refutation proofs [BBN11]. Quickcheck for example is a model checker that uses random model exploration and narrowing to find counter examples. Nitpick is a SAT based model checker used to generate finite models that satisfy the negation of the formula to refute.

Theorem provers mainly aim at helping to write machine checked proofs for theorems. They were used in many application to perform static verification proofs over programs. Theorem provers were also used in specification-based testing where sophisticated reasoning is needed. An example of a prover based testing framework is HOL-TestGen [BW12], and extension of Isabelle/HOL theorem prover. It supports test cases and test data generation from specifications expressed in higher order logics. Test selection hypotheses can be expressed explicitly in the system. Isabelle/HOL theories in addition to automated proof techniques are used in the test generation tactics. A test theorem is stated as a proof obligation, then test generation tactics are applied to simplify it in order to get test cases. HOL-TestGen takes advantage of the powerful symbolic computation and connected constraint solvers to Isabelle/HOL. An other combination of theorem proving and testing was explored by Burton et al. [BCM00]. The authors detailed different possible combinations of testing and proving, for instance a Z based test generation using CADiZ theorem prover. A similar application of test generation from Z specifications based on Isabelle/HOL theorem prover is the work of Helke et al. [HNS97]. Test generation in this method is based essentially on DNF decomposition using the reasoning facilities of the prover. Isabelle simplifier is also used in test generation in order to eliminate infeasible test cases.

## 2.4 Conclusions

This chapter gives a quick overview of the relation between formal methods and testing. The formal methods considered in this chapter are of two classes: formal specification languages and (traditional) formal verification methods. A third kind of formal methods is introduced in this chapter, which is formal testing methods. These provide a formal test environment containing test definitions and generation and result evaluation techniques (verdicts and oracles).

Some relations between formal testing and the other kinds of formal methods are introduced in this chapter. First, some formal specification methods are used as basis for some test definitions and generation techniques. These specification formalisms are used to describe data or behavioral aspects (or both) of systems. An other kind of interactions appears when some formal verification methods and techniques are deviated to be used for testing. Examples of this interaction are theorem prover and model checkers used for testing.

The next chapter will introduce the context of the formal specification and testing environment presented in this thesis. The context includes introductions to the formal specification language *Circus*. It also contains an introduction to the formal infrastructure of our work *i.e.* Isabelle/HOL theorem prover.

CHAPTER 3

# Context

## Contents



## 3.1 Introduction

The main goal of this thesis is to provide a formal environment for *Circus* on the basis of Isabelle/HOL. This chapter introduces the context and the basics of this environment: the *Circus* language and the Isabelle/HOL theorem prover. *Circus* is a well-defined formal specification language, covering several aspects of system

specifications. Isabelle/HOL is a powerful formal framework for logical reasoning and theorem proving and many other applications.

The next section introduces *Circus*, including its syntax, semantics, UTP foundations and refinement and testing theories. Section 3.3 gives an overview of Isabelle/HOL and some of its important notions used in this thesis.

## 3.2 The Circus Language

*Circus* [WC02] is a formal specification language which integrates the notions of states and complex data types in a Z-like style and communicating parallel processes inspired from CSP. From Z, the language inherits the notion of schema used to model sets of (ground) states as well as the ability to describe state modifications via preconditions and postconditions. From CSP, the language inherits the concept of *communication events* and typed communication channels, the concepts of deterministic and non-deterministic choice (reflected by the combinators $P \,\Box\, P'$ and $P \sqcap P'$), the concept of concealment (hiding) $P \backslash A$ of events in $A$ occurring in the evolution of process $P$. Due to the presence of state variables, the *Circus* parallel operator is slightly different from CSP. Its syntax is given as: $P \,[\![ n \mid c \mid n' ]\!]\, P'$ means that $P$ and $P'$ communicate via the channels mentioned in $c$; moreover, $P$ may modify the variables mentioned in $n$ only, and $P'$ in $n'$ only, $n$ and $n'$ being disjoint name sets. Moreover, the language comes with a formal notion of refinement based on a denotational semantics.

The language follows the failure/divergence semantics [RHB97], (but coined in terms of the UTP [OCW07]), providing a notion of execution trace `tr`, refusals `ref`, and divergences. It is expressed in terms of the UTP [HH98] which makes it amenable to other refinement-notions in UTP. The semantics allows for a rich set of algebraic rules for specifications and their transitions to program models.

A simple *Circus* specification of *FIG*, the fresh identifiers generator process, is given in Figure 3.1.

In this example, the type *ID* is first declared in a Z style type declaration. Channels are then defined, *req* for synchronization and *ret* and *out* to communicate values of type *ID*. After these global declarations, the definition of the process is introduced. It contains a state declaration as a schema expression and schema operations and actions that may manipulate the state. Finally, a main action is defined using the previously defined schema operations and actions.

### 3.2.1 Syntax

The syntax of *Circus* provides a rich collection of constructs that allows for describing complex specifications. It integrates the syntax of Z and CSP with new constructs

$[ID]$

**channel** $req$
**channel** $ret, out : ID$

**process** $FIG \mathrel{\widehat=}$ **begin**

**state** $S == [\, idS : \mathbb{P}\ ID \,]$

$Init \mathrel{\widehat=} idS := \emptyset$

*Out*

$\Delta S$
$v! : ID$

---

$v! \notin idS$
$idS' = idS \cup \{v!\}$

*Remove*

$\Delta S$
$x? : ID$

---

$idS' = idS \setminus \{x?\}$

• $Init$ ; **var** $v : ID$ •
$(\mu\ X \bullet (req \rightarrow Out\,;\ out!v \rightarrow Skip \ \Box\ ret?x \rightarrow Remove)\,;\ X)$

**end**

Figure 3.1: The Fresh Identifiers Generator in *Circus*

for notions specific to *Circus*. State declaration and manipulation is handled with Z schemas. Behavioral aspects are described using CSP operators.

The top most element of a *Circus* specification is a program. We introduce in the following the global structure of *Circus* programs based on its syntax [Oli06]. The complete BNF syntax of *Circus* can be found in Appendix A.1.

### Programs

The top most construct in a *Circus* specification is a program. A *Circus* program describes a possibly empty sequence of paragraphs in a Z style.

$$\mathsf{Program} \ ::= \ \mathsf{CircusPar}^*$$

Each *Circus* paragraph can either be a Z paragraph, a channel declaration, a channel set definition or a process declaration. The syntax of Z paragraphs is the same as given in [Spi92]. In the following, the syntactic category N stands for valid Z identifiers.

$$\mathsf{CircusPar} \ ::= \ \mathsf{Par} \mid \textbf{channel}\ \mathsf{CDecl} \mid \textbf{chanset}\ \mathsf{N} == \mathsf{CSExp} \mid \mathsf{ProcDecl}$$

In the *FIG* example given in Figure 3.1, the program is described with 4 paragraphs. First, a Z paragraph for type declaration, then two channel declaration paragraphs and finally, a process declaration paragraph.

### Channels and channel sets

Communication and synchronization channels must be declared in order to be used in the process description. The syntactic categories Exp and SchemaExp are Z expressions and Z schema expressions.

$$\begin{array}{lll} \mathsf{CDecl} & ::= & \mathsf{SimpleDecl} \mid \mathsf{SimpleDecl};\ \mathsf{CDecl} \\ \mathsf{SimpleDecl} & ::= & \mathsf{N}^+ \mid \mathsf{N}^+ : \mathsf{Exp} \mid [\mathsf{N}^+]\mathsf{N}^+ : \mathsf{Exp} \mid \mathsf{SchemaExp} \end{array}$$

A channel may either be a synchronization channel or a communication channel. A synchronization channel does not communicate any value. Its declaration is performed only by giving its name. For communication channels, the type of the communicated values must be associated with the channel name. Similar channels declarations can be grouped by giving a comma-separated list of names in the declaration. A channel declaration can also be given in a generic way, defining a family of channels. Finally, schema expressions (without predicate part) can be used to declare channels or groups of channels.

In the *FIG* example (Figure 3.1), two channels declarations are given. The first declaration introduces the synchronization channel *req*. The second declaration encloses two communication channels declarations *ret* and *out* having the same values type *ID*.

Channel sets are used to define a named set of channels that can be used in some CSP operators (*e.g.* parallel composition and hiding). A channel set declaration is performed by giving its name and a set of previously defined channels names. A channel set may either be an empty set, an enumeration of channels names or a combination of channel sets using some set operators.

$$\mathsf{CSExp} ::= \{|\}\ \mid \{|\ \mathsf{N}^+\ |\}\ \mid \mathsf{N} \mid \mathsf{CSExp} \setminus \mathsf{CSExp} \mid \mathsf{CSExp} \cup \mathsf{CSExp} \mid \mathsf{CSExp} \cap \mathsf{CSExp}$$

### Processes

The most interesting part of a *Circus* specification is process declaration paragraphs. A process declaration is composed of a process name and a process definition. Generic declaration of families of processes is also possible.

$$\mathsf{ProcDecl} ::= \mathbf{process}\ \mathsf{N} \mathrel{\widehat=} \mathsf{ProcDef} \mid \mathbf{process}\ \mathsf{N}[\mathsf{N}^+] \mathrel{\widehat=} \mathsf{ProcDef}$$

A process definition is a (possibly prefixed or parameterized) process. Each process can be given by an explicit definition or by combining other processes.

Explicit process definitions are delimited with two keywords **begin** and **end**. It is defined using a sequence of process paragraphs and a state declaration in terms of a Z schema expression. The definition contains also a nameless action definition describing the main action of the process.

$$\begin{array}{lcl} \mathsf{ProcDef} & ::= & \ldots \mid \mathsf{Proc} \\ \mathsf{Proc} & ::= & \textbf{begin}\ \mathsf{PPar}^*\ \textbf{state}\ \mathsf{SchemaExp}\ \mathsf{PPar}^*\ \bullet\ \mathsf{Action}\ \textbf{end} \\ & \mid & \ldots \end{array}$$

In the example of Figure 3.1, the process *FIG* is declared explicitly. It is composed of a state definition, three paragraphs and a main action.

### Actions

A process paragraph can either be a named (possibly parameterized) action or a variable names set definition. A named action declaration is given by its name and its definition. The name set declaration is very similar to channel set declaration, but with variable names instead of channel names.

$$\mathsf{PPar} ::= \mathsf{N} \mathrel{\widehat=} \mathsf{ParAction} \mid \textbf{nameset}\ \mathsf{N} == \mathsf{NSExp}$$

An action definition can either be simple or parameterized where the body is prefixed by some parameter declaration. An action can be a Z schema expression, a guarded command, an invocation of a previously defined action or a combination of action using CSP operators. An action can be defined from another action definition by renaming variables and state components. This part of the syntax gives a concrete view of the full integration of CSP, Z and guarded commands.

$$\begin{array}{lcl} \mathsf{ParAction} & ::= & \mathsf{Action} \mid \mathsf{Decl} \bullet \mathsf{ParAction} \\ \mathsf{Action} & ::= & \mathsf{SchemaExp} \mid \mathsf{Command} \mid \mathsf{N} \mid \mathsf{CSPAction} \mid \mathsf{Action}[\mathsf{N}^+ := \mathsf{Exp}^+] \end{array}$$

Commands may also be defined using Dijkstra's guarded commands [Dij76]. This category contains, among other commands, assignments, guarded alternations, variable blocks (scopes) and specification statements.

$$\begin{array}{lcl} \mathsf{Command} & ::= & \mathsf{N}^+ := \mathsf{Exp}^+ \mid \textbf{if}\ \mathsf{GActions}\ \textbf{fi} \mid \textbf{var}\ \mathsf{Decl} \bullet \mathsf{Action} \mid \mathsf{N}^+ : [\mathsf{Pred}, \mathsf{Pred}] \\ & \mid & \ldots \end{array}$$

CSP operators are used to combine *Circus* actions. Their syntax is the same as in [RHB97] except for parallel composition and interleaving. Basic CSP actions are *Stop*, *Skip* and *Chaos*. Examples of CSP operators are prefixed and guarded actions, sequential and parallel compositions, internal and external choices and hiding. As said above, parallel composition requires more elements to deal with local data updates. Two distinct variable name sets are associated to each parallel action.

Each action has write permissions in the global state for variables appearing in his name set.

$$
\begin{array}{rcl}
\mathsf{CSPAction} & ::= & \mathit{Stop} \mid \mathit{Skip} \mid \mathit{Chaos} \mid \mathsf{Comm} \rightarrow \mathsf{Action} \mid \mathsf{Pred} \,\&\, \mathsf{Action} \\
& \mid & \mathsf{Action};\ \mathsf{Action} \mid \mathsf{Action} \,\Box\, \mathsf{Action} \mid \mathsf{Action} \sqcap \mathsf{Action} \\
& \mid & \mathsf{Action} \,[\![\, \mathsf{NSExp} \mid \mathsf{CSExp} \mid \mathsf{NSExp} \,]\!]\, \mathsf{Action} \mid \mathsf{Action} \setminus \mathsf{CSExpression} \\
& \mid & \ldots
\end{array}
$$

Actions can be prefixed by a communication. A communication is either a synchronization, an output, a simple or constrained input or a multi directional communication.

The *FIG* example of Figure 3.1 is defined using sequential composition, variable scoping, recursion, prefixing and external choice operators. Prefixed actions includes a synchronization, an input and an output communications.

In the context of this thesis only a basic (shorter) version of this syntax is considered. The short BNF syntax of the studied version of *Circus* is also presented in Appendix A.2. Our choice can be justified in two points. First, the considered syntax covers the most important aspects of the language. Number of the omitted operators can be defined from these basic ones. Another reason of this choice is that our proposed representation is a shallow embedding *i.e.* relies on semantics and not on syntax. Thus, *Circus* operators and actions are defined independently from each others. Consequently, the omitted operators can be added to the system without altering its global consistency. In fact, adding a new action or operator definition does not require any modifications on the existing operators.

### 3.2.2 Semantics

*Circus* is defined semantically using a denotational and an operational semantics. The basis of these semantics is the Unifying Theories of Programming (UTP) introduced by Hoare and He [HH98]. Both data and behavioral aspects are represented in the semantics in a unified way. This makes it possible to reason about *Circus* specifications as entire entities. We introduce in the sequel the basic notions of UTP followed by the denotational and operational semantics of *Circus*.

#### 3.2.2.1 UTP

The Unifying Theories of Programming (UTP) is a semantic framework based on an alphabetized relational calculus. It provides a theory of relations that can be used to unify many different languages paradigms. Every theory is represented in UTP by a set of variable names called alphabet with some relations defined over initial and intermediate or final observations (values) of variables.

Common notions such as sequential and parallel compositions, conditionals and nondeterminism are used in different paradigms and formalisms. UTP provides a unified representation of these notions in terms of predicates and relations. Sequential composition can be seen for example as relational composition, conditional as a predicate, nondeterminism as disjunction and parallel composition is a sort of conjunction. Other notions related to processes like traces, refusals and termination are sketched in some particular forms of UTP relations.

The unification of all these notions in one framework makes it possible to address different paradigms in a unified way. It offers a basis for reasoning about different languages and greatly facilitates their study and comparison. Moreover, such a unification offers means of combining different languages describing various facets and artifacts of software development in a seamless, logically consistent way. This is the case of *Circus* which is based essentially on a combination of Z and CSP.

The properties of a language represented in UTP can be expressed in terms of healthiness conditions. Healthiness conditions restrict the space of predicates or relations considered for this language. UTP contains definition for healthiness conditions that characterize some common notions *e.g.* designs and reactive processes. UTP healthiness conditions are summarized in Appendix B.1.2. The different theories defined in UTP are presented in Figure 3.2, we briefly introduce them in the sequel.

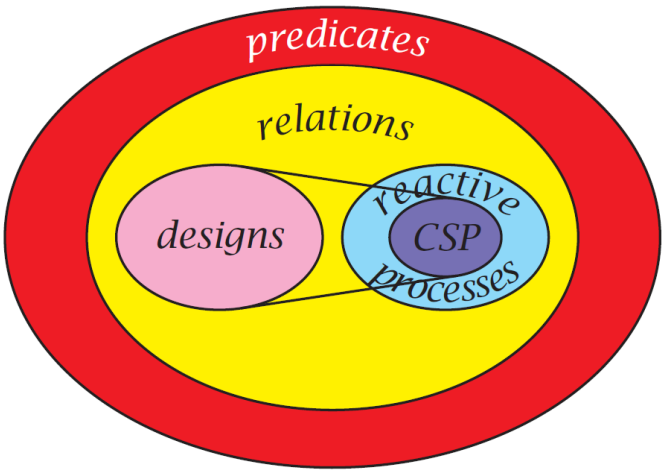



Figure 3.2: UTP theories

**Alphabetized predicates** The most general and basic theory of the UTP is the theory of alphabetized predicates. They are represented as pairs $(\alpha P, P)$, where $P$ is a predicate defined over variables whose names appear in the alphabet $\alpha P$. The

alphabet collects the names of observational variables used to describe systems in terms of predicates. Logical connectives (*e.g.* conjunction, disjunction ... *etc*) are defined for alphabetized predicates.

**Alphabetized relations** Special cases of predicates are relations with variables that can be observed in an initial and in an intermediate or final state. These observations are denoted respectively by undecorated and dashed variable names. A relation alphabet $\alpha P$ is composed of an input alphabet $in\alpha P$ and an output alphabet $out\alpha P$. The UTP theory of relation introduces some common notions of relational calculus like assignment, composition, conditional and variable scoping. Nondeterminism is captured using relational disjunction and recursion by the weakest fixed point operator defined for the complete lattice of relations.

**Designs** In order to characterize a subclass of relations that are more suitable for program designs, a special observational variable is considered. This variable, called $ok$, characterizes with its initial value the fact that the program has already started. The final observation of this variable *i.e.* $ok'$ indicates if the program has successfully terminated its execution. The program is then described in forms of pre- post-conditions and a design is written: $(P \vdash Q)$. This expression is defined by the relation $ok \wedge P \rightarrow ok' \wedge Q$ which means the following: if a program started its execution with a precondition $P$, at the end of its execution it will satisfy the postcondition $Q$. Another way to characterize relations that are designs is the use of 4 predefined healthiness conditions. An example of designs healthiness conditions is **H2** defined by: "$P(A, A'[false/ok]) \rightarrow P(A, A'[true/ok])$" which means that a design may not require non-termination. A list of all designs healthiness conditions is given in Appendix B.1.2.

**Reactive processes** Relations that describe behavioral aspects of reactive processes are characterized by adding other observational variables and healthiness conditions. The variable *wait* indicates if the output state of a reactive process is an intermediate or a final state. The second variable of reactive processes is $tr$, recording the trace of events already performed by the process. The last observational variable of reactive processes is $ref$, representing the set of events the process may refuse at a given state. Three healthiness conditions (**R1**, **R2** and **R3**) are defined to characterize the properties of reactive processes. Usually, a functional composition of these three healthiness condition in one condition called **R** is used to characterize reactive processes. The definitions of theses healthiness conditions are given in Appendix B.1.2.

**CSP processes** A particular class of reactive processes is the class of CSP processes, characterized by adding two other healthiness conditions (**CSP1** and **CSP2**).

These healthiness conditions are defined in Appendix B.1.2. A denotational semantics of CSP basic processes and operators is also given in terms of reactive processes.

**Circus actions** In UTP designs programs are represented in a pre- post-condition style. Behavioral aspects of CSP processes are captured in the theory of reactive processes. *Circus* actions are defined in a unified theory based on these two aspects. CSP healthy reactive designs are then used to characterize the class of *Circus* actions. In addition to this characterization, other healthiness conditions are defined for *Circus*, in order to capture some particular properties.

The complete list of UTP healthiness conditions is given in Appendix B.1.2.

#### 3.2.2.2 Denotational semantics

The first definition of the denotational semantics of *Circus* was based directly on Z [WC02]. This definition allowed to reason about particular *Circus* actions and their properties as Z specifications. This representation restricted *Circus* processes to a particular form and did not allow to prove meta theorems on the language. Another definition of the denotational semantics was proposed [OCW07]. It was inspired from the first definition and based on UTP.

This denotational semantics of *Circus* is given in terms of UTP reactive designs. The semantics of all actions and operators is given in Appendix B.2. An example of a *Circus* action semantics is given in the following:

$$Skip \mathrel{\widehat{=}} \mathbf{R}(true \vdash tr' = tr \land \neg\, wait' \land v' = v)$$

This can be read as a reactive design whose precondition is *true*. The postcondition states that the trace and the system variables $v$ are not changed, and that the system is in a final state.

In this version of the denotational semantics, all the actions are defined implicitly as reactive designs. This makes actions satisfy the healthiness conditions of reactive processes (**R1**-**R3**) and of CSP processes (**CSP1**-**CSP2**). In addition, *Circus* actions satisfy three other healthiness conditions: **C1**, **C2** and **C3**, which are given in Appendix B.1.2.

#### 3.2.2.3 Operational semantics

A Plotkin-style operational semantics for Circus was presented in [WCGF07, CG11]. The transition relation is also based on UTP and on the denotational semantics of *Circus*. It is defined over symbolic configurations formed by triples of the form:

$$(c \mid s \models A)$$

where $c$ is a constraint (a UTP condition) over the symbolic variables in use, $s$ a state *i.e.* an assignment of symbolic values to all *Circus* variables in the scope (a UTP condition over output variables), and $A$ is a *Circus* action.
The transition relation over configurations is of the form:

$$(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)$$

where $e$ is a label *i.e.* an event (channel $\times$ symbolic variable) or $\varepsilon$. If the label is an event, this means that the process needs to perform a communication in order to continue is execution. If the transition is labeled with $\varepsilon$ this means that the process can evolve silently.

The operational semantics is defined by a set of rules defining this transition relation for each *Circus* action. The semantics of the transition relation is defined using the UTP refinement relation. An example of a transition rule for assignment is given in the following:

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge (s;\ w_0 = e) \mid s;\ v := w_0 \models \mathit{Skip})}$$

This defines an internal action represented by a silent transition that updates the state and ends its execution. The symbolic variable $w_0$ is inserted to represent the assigned value in the state and in the constraint. The UTP relational composition operator " ; " is used to compose the state with an assignment relation, this is equivalent to a state update operation.

All the rules of the operational semantics are presented in Appendix C.2.

### 3.2.3 Refinement and testing

*Circus* specifications and semantics were used as a basis of two substantial applications: refinement and testing. In a formal system specification and development approach, refinement is a very important notion. Abstract specifications are first written. Then intermediate more concrete specifications are obtained by successive refinement from the initial one. The last step is to generate, if possible, the implementation from the most refined specification. In other cases (*e.g.* the implementation cannot be generated), the last step will be to check that the realized system is a refinement of the last specification. Refinement proofs are rarely possible at this stage for different reasons. The verification can be done by testing the refinement relation on the concrete system.

In this thesis, we provide our formalizations of these two important applications using the semantics of *Circus*. In Chapter 4 we show how to assist refinement proofs

on *Circus* processes using the Isabelle/HOL proof assistant. In Chapter 5 we present a test generation procedure for refinement based on *Circus*.

#### 3.2.3.1 Refinement

In [SWC02] the notion of refinement is defined for *Circus* processes and actions. This definition covers both data oriented and behavioral aspects of the language. In UTP, the refinement relation is defined by a generalized inverted implication as follows:

$$[P \Rightarrow S]$$

which means that the specification $S$ is refined by the program $P$.

Refinement of *Circus* actions is defined using the UTP refinement relation. An action $A_2$ is a refinement of an action $A_1$ means that their respective denotational semantics satisfy the UTP refinement relation. This refinement relation is defined as follows:

$$A_1 \sqsubseteq_A A_2 \equiv [A_2 \Rightarrow A_1]$$

This definition is only valid when the two actions $A_1$ and $A_2$ are defined in the same state space. In a general refinement activity, the state can be refined too in the same way as variable blocks, and modules can be data refined in imperative programs. *Circus* data refinement is based on the well-known forwards and backwards simulation. In *Circus*, a simulation is an abstraction relation between the states of two processes that satisfies some properties.

**Forwards simulation** If the simulation relation is defined from the abstract state to the concrete state, it is called a forwards simulation. Forwards means that the relation follows the same direction as the refinement relation. The definition of forwards simulation is given by the following:

**Definition 3.2.1.** *A forwards simulation between actions $A_1$ and $A_2$ of processes $P_1$ and $P_2$, with local state $L$, is a relation $R$ between $P_1.st$ , $P_2.st$ , and $L$, satisfying*

1. *(feasibility)* $\forall P_2.st \; L \bullet (\exists P_1.st \bullet R)$
2. *(correctness)* $\forall P_1.st \; P_2.st \; P_2.st' \; L \bullet R \wedge A_2 \Rightarrow (\exists P_1.st' \; L' \bullet R' \wedge A_1)$

*were $P.st$ represents the input state of the process $P$ and $s'$ is the output state containing the dashed version of variables in $s$.*

*In this case, we write $A_1 \preceq_{P_1,P_2,R,L} A_2$ and say that the action $A_2$ simulates the action $A_1$, according to the simulation $R$, and in a state extended by $L$. When clear from the context, we omit the subscripts. A forwards simulation between $P_1$ and $P_2$ is a forwards simulation between their main actions.*

In addition to this definition, a theorem proving its soundness and some general simulation laws are provided. The backwards simulation can be seen as a dual of forwards simulation.

**Backwards simulation** The simulation relation can be defined as an abstraction of the concrete state to the abstract state. In this case, it is called a backwards simulation since it follows the opposite direction of the refinement relation. Backwards simulation is defined by the following:

**Definition 3.2.2.** *A backwards simulation between actions $A_1$ and $A_2$ of processes $P_1$ and $P_2$, with local state $L$, is a relation $R$ between $P_1.st$ , $P_2.st$ , and $L$, satisfying*

1. *(feasibility)* $\forall P_2.st \; L \bullet (\exists P_1.st \bullet R)$
2. *(correctness)* $\forall P_1.st' \; P_2.st \; P_2.st' \; L' \bullet R' \land A_2 \Rightarrow (\exists P_1.st \; L \bullet R \land A_1)$

*In this case, we write $A_1 \preceq_{P_1,P_2,R,L} A_2$ and say that the action $A_2$ simulates the action $A_1$, according to the simulation $R$, and in a state extended by $L$. When clear from the context, we omit the subscripts. A backwards simulation between $P_1$ and $P_2$ is a backwards simulation between their main actions.*

**Simulation laws** Some general refinement laws are defined and proved for the distributivity of the simulation relation on *Circus* operators. Basic actions *Skip*, *Stop* and *Chaos* are independent from the state. Consequently, they are not affected by the simulation relation *e.g. Skip* $\preceq$ *Skip* for any simulation relation.

For other operators, that may be affected by the simulation relation, refinement laws are defined. The following example describe the forwards simulation rule for schema expressions, which is very similar to the standard Z rule.

**Law 3.1.**

$$ASExp \preceq CSExp$$

***provided***

- $\forall P_1.st; \; P_2.st; \; L \bullet R \land \text{pre } ASExp \Rightarrow \text{pre } CSExp$
- $\forall P_1.st; \; P_2.st; \; P_2.st'; \; L \bullet R \land \text{pre } ASExp \land CSExp \Rightarrow (\exists P_1.st'; \; L' \bullet R' \land ASExp)$

The law contains an applicability condition in addition to the general simulation relation definition. The applicability condition restricts the simulation relation to the sub-domain defined by the schema's precondition. Another example is the input prefixing operator, whose law is defined as follows:

**Law 3.2.**

$$c?x \rightarrow A_1 \preceq c?x \rightarrow A_2$$

***provided*** $A_1 \preceq A_2$

All defined and proved simulation laws can be found in [Oli06].

#### 3.2.3.2 Testing for refinement

In [CG11] the foundations of testing based on *Circus* specifications are stated for two conformance relations: traces refinement and deadlocks reduction (usually called *conf* in the research community of test derivation from transition systems). The basis of this work is the operational semantics that expresses in a symbolic way the evolution of systems specified in *Circus*.

Using this operational semantics, symbolic characterizations of traces, initials, and acceptance sets have been stated and used to define relevant notions of tests. Two symbolic exhaustive test sets have been defined respectively for traces refinement and deadlocks reduction: the proofs of exhaustivity guarantee that, under some basic testability hypotheses, a system under test (SUT) that would pass all the concrete tests obtained by instantiation of the symbolic tests of the symbolic exhaustive test set satisfies the corresponding conformance relation.

The tests are defined using the following notions:

- *cstraces* : a constrained symbolic trace is a pair formed by a symbolic trace $st$ and a constraint $c$. A symbolic trace is a finite sequence of symbolic events $d!\alpha_0$ or $d?\alpha_0$ or $d$, where $d$ is a channel, and $\alpha_0$ is a symbolic variable that represents the value communicated. The constraint $c$ is a predicate over the symbolic variables used in $st$. These symbolic variables are different from the variables used in the state of any *Circus* model. They are names used to represent communicated values and the dependencies between them (they are similar to the indexed symbolic values used during symbolic program execution).

- *csinitials*: the set *csinitials* associated with a constrained symbolic trace $(st, c)$ of a *Circus* process $P$ contains the constrained symbolic events that represent valid continuations of $(st, c)$ in $P$. Constrained symbolic events are pairs formed by a symbolic event, and a constraint that refers to the symbolic variables of the event, as well as those occurring in $st$.

- $\overline{csinitials}$ : given a process $P$ and one of its constrained symbolic traces $(st, c)$, the set $\overline{csinitials}$ contains the constrained symbolic events that represent the events that are not initials of $P$ for any of the instances of $(st, c)$.

- *csacceptances*: a *csacceptances* set associated with a constrained symbolic trace $(st, c)$ of a *Circus* process $P$ is a set of sets $SX$ of symbolic acceptances. In the context of a constraint $c_1$ and a state $s_1$ after the trace $(st, c)$, we consider all stable configurations that can be reached from it. For each of them, we require $SX$ to include at least one element of its *csinitials*. A stable configuration is one from which there are no available silent transitions.

These notions are illustrated in the following example for the *FIG* process introduced in 3.1:

$$
\begin{array}{ll}
cstrace & ([req, out!a, req], true) \\
\underline{csinitials} & \{(out!b, a \neq b)\} \\
\overline{csinitials} & \{(out!b, a = b), (req, true), (ret?b, true)\} \\
csacceptances & contains\ all\ the\ supersets\ of \{(out!b, a \neq b)\}
\end{array}
\qquad (\clubsuit)
$$

**Symbolic tests for traces refinement** Traces refinement corresponds to trace inclusion: process $P_2$ is a traces refinement of process $P_1$ if and only if the set of traces of $P_2$ is included in that of $P_1$. The definition of symbolic tests for traces refinement is based on a constrained symbolic trace *cst* of the *Circus* process $P$ used to build the tests, followed by a forbidden symbolic continuation, namely a constrained symbolic event *cse* belonging to the set $\overline{csinitials}$ associated with *cst* in $P$. The goal of such tests is to provoke the execution of a trace that is not a trace of the specification. Some extra events *pass*, *fail*, and *inc* are used for verdicts. They must not be used in the trace or in the forbidden continuation, or more generally, in $P$. The symbolic test is the sequence of the symbolic events of *cst* and then *cse*, interspersed with one of the three verdict events as follows:

- *pass* in inserted between the last event of *cst* and *cse*, since after *cst* the system under test must not accept the forbidden continuation and deadlock;
- accordingly, *fail* is inserted after *cse*, since *cse* must not be accepted;
- *inc* is inserted before *cst* and between the events of *cst*, since if a strict prefix followed by a deadlock is observed, the trace of $P$ has not been executed by the SUT, and this is quite acceptable from the definition of traces refinement; the test execution is said to be *inconclusive*;

Thus, if the last event of an execution of an instance of the symbolic test is *pass*, then the SUT passes the test; similarly, if the last event is *fail*, then the SUT fails the test. Finally, if the last event is *inc*, then the test is inconclusive.

From the example ♣, the resulting symbolic tests are:

$inc \rightarrow req \rightarrow inc \rightarrow out?a \rightarrow inc \rightarrow req \rightarrow pass \rightarrow out?b : (a = b) \rightarrow fail \rightarrow Stop$
$inc \rightarrow req \rightarrow inc \rightarrow out?a \rightarrow inc \rightarrow req \rightarrow pass \rightarrow req \rightarrow fail \rightarrow Stop$
$inc \rightarrow req \rightarrow inc \rightarrow out?a \rightarrow inc \rightarrow req \rightarrow pass \rightarrow ret!b \rightarrow fail \rightarrow Stop$

Given a *Circus* process $P$ the set of all the symbolic tests described above is a symbolic exhaustive test set with respect to traces refinement: a SUT that would pass all the instances of all the symbolic tests is a traces refinement of $P$, assuming some basic testability hypotheses that are given in [CG11].

**Symbolic tests for deadlocks reduction** Deadlocks reduction (also called *conf*) requires that deadlocks of process $P_2$ are deadlocks of process $P_1$. Actually, the notion of failures refinement in CSP is the conjunction of traces refinement and *conf*, as shown in [CG07]. Similarly to what was done above for traces refinement, the definition of symbolic tests for deadlocks reduction is based on a constrained symbolic trace *cst* of $P$ followed by a choice over a set $SX$, which is a symbolic acceptance of *cst* in $P$. Verdict events are inserted in a different way as for traces refinement, since after performing an instance of *cst* the SUT must accept some event in the proposed choice an must not deadlock. Namely:

- *pass* is inserted after every event of $SX$;
- *fail* is inserted after *cst*, since some event of $SX$ must be accepted;
- *inc* is inserted before and between the events of *cst* as for traces refinement.

As above, the last verdict event performed when executing an instance of the symbolic test yields the verdict of the test experiment.

From the example ♣, one resulting symbolic tests may be:

$$inc \rightarrow req \rightarrow inc \rightarrow out?a \rightarrow inc \rightarrow req \rightarrow fail \rightarrow \\ (out?b : (a \neq b) \rightarrow pass \rightarrow Stop \,\Box\, ret!b \rightarrow pass \rightarrow Stop)$$

Given a *Circus* process $P$, the set of all the symbolic tests described above is a symbolic exhaustive test set with respect to deadlocks reduction [CG11].

**From symbolic to concrete tests** The test sets defined previously are symbolic since they are based on symbolic traces initials and acceptances. In practice, tests must be defined on concrete values in order to be executed. Instantiation functions are defined to instantiate symbolic events and traces to concrete ones. These functions are the basis for other test instantiation functions, returning for a symbolic test, the set of corresponding concrete tests.

Constraint solving techniques, associated to some selection hypotheses, can be used for test instantiation. This can either be done off-line (*i.e.* before test execution) or on-line (*i.e.* associated to test execution). Off-line instantiation is simpler to realize: a concrete test is produced by constraint solving over all symbolic values. This concrete test is then executed directly on the system. The problem with this

approach is that the system can produce an output that satisfies the original constraint of the symbolic test but different from the value produced in the concrete test. A solution of this problem is to use an on-line instantiation approach.

In on-line test instantiation, concrete values are produced only for system inputs, outputs are kept symbolic. Values that are returned by the system when executing the test are used to instantiate remaining values in the test. The definition of the test is slightly changed to take into account the case of output values that does not satisfy the associated constraint.

## 3.3 Isabelle/HOL

Isabelle/HOL is a formal logical framework used as a basis for several formal environments. All the theories and formal developments of this thesis are built on top of it. Besides Isabelle/HOL, a number of logical frameworks are used for the formalization of mathematical theories and proofs. Systems like HOL [HOL], Coq [Coq], PVS [PVS] and ProofPower [PP] are good examples. All these systems offer a formal basis for specification and reasoning but with (more or less important) differences.

Among all the available systems, we made the choice to use Isabelle/HOL as basis of our work for different reasons. First, Isabelle/HOL is one of the most powerful and rich proof assistants. It offers a large number of library theories that can be reused for particular applications. It also integrates several external tools (*e.g.* constraint solvers and model checkers) and supports code generation. Isabelle/HOL counts with a very large community of users and developers that keep it highly maintained and supported. Another reason is that our work was inspired from some existing applications based on Isabelle/HOL, *e.g.* HOL-Z, HOL-CSP and, last but not least, HOL-TesGen which is a formal test-generation framework.

### 3.3.1 Isabelle, HOL and Isabelle/HOL

#### 3.3.1.1 Isabelle

Isabelle [NPW02] is a generic theorem prover implemented in SML. It is based on the "LCF-style architecture", which makes it possible to extend a small trusted logical kernel by user-programmed procedures in a logically safe way. New object logics can be introduced to Isabelle by specifying their syntax and semantics. The inference rules of these logics can be derived and specific tactic support for the object logic can be added. Isabelle is based on a typed $\lambda$-calculus including a Haskell-style type-system including type-classes *e.g.* in $'\alpha$`::order`, the type-variable ranges over all types that possess a partial ordering.

The meta logic of Isabelle [Pau89], provided in the Isabelle/Pure framework, is a minimal higher order logic. Different object logics (*e.g.* HOL) are formalized by extending this meta logic. It defines three (meta) connectives: $\bigwedge$ for universal quantification, $\Longrightarrow$ for implication and $\equiv$ for equality. These connectives are used to introduce inference rules and definitions of an object logic.

For example, the following introduction rules for conjunction and universal quantification:

$$\frac{P}{\forall x.P}\ (\text{allI}) \qquad \frac{P \quad Q}{P \wedge Q}\ (\text{conjI})$$

are expressed in the meta logic as follows:

```
1 (⋀ x. P x) ⟹ ∀ x. P x                    ⟦P; Q⟧ ⟹ P ∧ Q
```

where the brackets $[\![ \ldots ]\!]$ delimits premises of a rule and the implication $\Longrightarrow$ separates the premises from the conclusion.

#### 3.3.1.2 Higher-order logic (HOL)

HOL [Chu40] is a classical logic based on a simple type system. It provides the usual logical connectives like $\wedge$, $\rightarrow$ and $\neg$ as well as the object-logical quantifiers $\forall x \bullet P\,x$ and $\exists x \bullet P\,x$. In contrast to first-order logic, quantifiers may range over arbitrary types, including total functions $f :: \alpha \Rightarrow \beta$. HOL is centered 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 some polymorphically typed $\lambda$-calculus, HOL can be viewed as a combination of a functional programming language like SML or Haskell and a specification language providing powerful set notations and logical quantifiers ranging over elementary and function types.

#### 3.3.1.3 Isabelle/HOL

Isabelle/HOL is an instance of Isabelle with higher order logic. It provides a rich collection of library theories like sets, pairs, relations, partial functions lists, multisets, orderings, and various arithmetic theories which only contain rules derived from conservative, *i.e.* logically safe definitions. Setups for the automated proof procedures like `simp`, `auto`, and arithmetic types such as `int` are provided.

We introduce in the sequel some basic notions of Isabelle/HOL, including types, logical connectives, sets, pairs and lists. More advanced constructs of Isabelle/HOL will be introduced in section 3.3.2.

**Types** HOL is a typed logic with a type system similar to that of functional programming languages. Isabelle provides a powerful static type-checker, so only

well-typed terms can be expressed. In Isabelle/HOL, 4 type categories can be distinguished:

*base types* predefined in the library, including `bool` the type of truth values, `nat` the type of natural numbers and `int` the type of integers.

*type constructors* used to define generic types, arguments are associated to the constructor in a postfix style. Popular type constructors are `list` and `set` that can be used to define for example `nat list` or `nat set` for a list or a set of natural numbers.

*function types* noted $\Rightarrow$, *e.g.* `nat` $\Rightarrow$`bool` is the type of predicates over natural numbers. This definition covers only total functions types, partial functions are defined using total functions returning optional values.

*type variables* written `'a`, `'b` ... and used to define polymorphic types *e.g.* `'a` $\Rightarrow$ `bool` for the type of all the predicates.

Isabelle/HOL is strongly typed thus all terms and formulas must be well-typed in order to be correctly parsed. Generic types can be instantiated implicitly by the system using type inference *e.g.* when applying a function of a concrete type to an argument of a generic type.

**Logical connectives** The usual logical connectives $\wedge$, $\vee$, $\neg$ and $\longrightarrow$ as well as quantifiers $\exists$ and $\forall$ are defined in Isabelle/HOL. The implication and the universal quantification are equivalent to those defined in the meta logic *i.e.* $\Longrightarrow$ and $\bigwedge$. The scope of HOL quantifiers never extends over meta connectives since HOL formulas are atomic objects of the meta logic. The logical equivalence is defined using the polymorphic HOL equality but more often using the object equality of HOL (which reflects meta equality).

Conditional is also defined for arbitrary types and can be used with its classical mix-fixed form `if _ then _ else _`. A more sophisticated case splitting operator is also available and can be used on defined data types. It can be used a follows: `case e of pattern1 => e1 | pattern2 => e2 ....`

**Sets** HOL defines a generic type `'a set` for sets of elements of type `'a`. Associated to this type, the usual set constructs and operators are defined, for instance $\in$, $\cup$, $\cap$ and $\subseteq$. The empty set is denoted by `{}` and the universal set b `UNIV`. A set can be defined by enumerating its elements *e.g.* `{a, b}` or by using the set comprehension *e.g.* `{x. P x}`. Generalized union and intersection operators can be expressed on sets *e.g.* $\bigcup$`a`$\in$`A.B`, $\bigcap$`a.B` (which is an abbreviation of $\bigcap$`a`$\in$`UNIV.B`) and $\bigcup$`B` for union over a set of sets..

**Pairs** Another important type defined in Isabelle/HOL is the type of pairs whose constructor is written `'a × 'b`. Two standard projection functions are defined to retrieve the first and the second element of a pair *i.e.* `fst` and `snd`. The generic definition of pairs type makes it possible to defined nested pairs. For example, the tuple `(a, b, c)` is defined as a pair nested to the right `(a, (b, c))`. This idea is used also in the definition of extensible records where each record field corresponds to an element of the tuple. In order to distinguish the fields of the record, some selectors are associated to positions in order to identify directly each element.

**Lists** An example of inductively defined data types in Isabelle/HOL is the generic list type `'a list`. It is defined using two constructors `[]` for an empty list and an infix constructor `#`. All lists are defined using these constructors *e.g.* the list `[a, b, c]` is an abbreviation of `a#b#c#[]`. A number of useful operators are defined for lists *e.g.* `@` for infix concatenation, `rev` for the reverse and `length`.

The inductive definition of lists allows for inductive reasoning over them. This is done using the following induction rule generated with the definition of list type:

```
1 P []  ⟹ (⋀a l. P l ⟹ P a#l) ⟹ P l
```

This rule can be used in inductive proofs to split the general predicate `P` over the list `l` in two distinct cases. The first case is the basic step where the predicate is applied to the empty list. The second case is the induction step defined using the list constructor. A similar induction rule is automatically proved with every inductive data type definition.

## 3.3.2 Advanced constructs in Isabelle/HOL

### 3.3.2.1 Constant definitions

In its easiest form, constant definitions are definitional logical axioms of the form $c \equiv E$ where c is a fresh constant symbol not occurring in $E$ which is closed (both *w.r.t.* variables and type variables). For example:

```
1 definition upd::(α⇒β)⇒α⇒β⇒(α⇒β)     ("_(_ := _)")
2 where      upd f x v ≡ λ z. if x=z then v else f z
```

This definition introduces a constant `upd` with its given type and produces the axiom `upd_def` that represents its definition. The pragma `("_(_ := _)")` for the Isabelle syntax engine introduces the notation `f(x:=y)` for `upd f x y`.

Moreover, an elaborate preprocessing allows for recursive definitions, provided that a termination ordering can be established; such recursive definitions are thus internally reduced to definitional axioms.

#### 3.3.2.2 Type definitions

Types can be introduced in Isabelle/HOL in different ways. The most general way to safely introduce new types is the type definition using `typedef` construct. This allows one to introduce a type as a non-empty subset of an existing type. More precisely, the new type is specified to be isomorphic to this non-empty subset. For instance:

```
typedef mytype = "{x::nat. x < 10}"
```

This definition requires that the set is non-empty: ∃`x. x∈{x::nat. x<10}`, which is easy to prove in this case:

```
by (rule_tac x = 1 in exI, simp)
```

where ***rule_tac*** is a tactic that applies an introduction rule and `exI` corresponds to the introduction of the existential quantification.

In a similar way, the `datatype` command allows one to define inductive data types. This command introduces a data type using a list of *constructors*. For instance, a logical compiler is invoked for the following introduction of the type `option`:

```
datatype α option  = None | Some α
```

which generates the underlying type definition and derives distinctness rules and induction principles. Besides the *constructors* `None` and `Some`, the following match-operator and his rules are also generated: case $x$ of None ⇒ ... | Some $a$ ⇒ ...

#### 3.3.2.3 Type classes

A type class describes a set of types satisfying a common interface. This notion is very similar to the notion of interfaces or abstract classes in object-oriented languages. It is inspired on Haskell type classes. In addition to the interface of the class described by its operations, Isabelle type classes describe general types properties (class axioms). Any instance of a given class must overload (implement) its operations and satisfy its properties. For example, one can define a type class `eq` for types that supports equality as follows:

```
class eq =
  fixes eq :: 'α ⇒'α ⇒bool
  assumes refl: eq a a
```

This command defines a type class `eq` that contains an operation `eq` that must be reflexive. In order to prove that a given type instantiates this class one should provide a definition for the `eq` operation and prove its reflexivity. General theorems can be proved on type class properties, they can be used for any type instantiating this class.

Moreover, Isabelle provides an inheritance mechanism for type classes, allowing them to be organized hierarchically. All subclasses of a given class inherit its operations and satisfy its axioms.

#### 3.3.2.4 Extensible records

Isabelle/HOL's support for *extensible records* is of particular importance for our work. Record types are denoted, for example, by:

```
record T = a::T1
           b::T2
```

which implicitly introduces the record constructor (|a:=e1,b:=e2|) and the update of record r in field a, written as r(|a:= x|). Extensible records are represented internally by cartesian products with an implicit free component $\delta$, *i.e.* in this case by a triple of the type $\mathtt{T}_1 \times \mathtt{T}_2 \times \delta$. The third component can be referenced by a special selector `more` available on extensible records. Thus, the record `T` can be extended later on using the syntax:

```
record ET = T + c::T3
```

The key point is that theorems can be established, once and for all, on `T` types, even if future parts of the record are not yet known, and reused in the later definition and proofs over `ET`-values. Using this feature, we can model the effect of defining the alphabet of UTP processes incrementally while maintaining the full expressivity of HOL *w.r.t.* the types of $\mathtt{T}_1$, $\mathtt{T}_2$ and $\mathtt{T}_3$.

### 3.3.3 HOL-Z, HOL-CSP and HOL-TestGen

**HOL-Z** [BRW03] is a specification and proof environment for Z built on top of Isabelle/HOL. It is based on a shallow-embedding of Z semantics in HOL. The key idea of this embedding is the encoding of schema operations as predicates. A proof environment for the Z schema calculus is developed on top of this embedding. HOL-Z allows for importing Z specifications written in LaTeX and type-checked by the ZeTa System. More recently, an integration of HOL-Z with the well-know CZT parser and type-checker was developed.

The embedding, presented in this thesis for *Circus* and the UTP, in Isabelle/HOL is inspired by the shallow embedding of Z in HOL. However, HOL-Z cannot be used directly in *Circus* since the underlying semantics encoding of Z is different and in particular the state representation. The state representation in Isabelle/*Circus* is more modern and relies on predefined theories of Isabelle/HOL. We believe, nevertheless, that several conceptual ideas behind HOL-Z can be reused and adapted to fit with our work.

**HOL-CSP** [TW97] is a shallow embedding of CSP failures/divergences semantics into HOL. The CSP process type is first characterized using the well formedness rules of CSP. The different operators are then defined by embedding their semantics in the process type and then by proving that they are well formed. The embedding instantiates some predefined type classes (*e.g.* ordering) and reuses their inherited notions and properties (*e.g.* fix-point operator). HOL-CSP provides a basis for reasoning about CSP specifications using the CSP denotational semantics.

The semantics of CSP operators defined in *Circus* is based on the failures/divergences model but expressed in terms of UTP. The characterization of the process type and the representation of communication events and traces inspired in a significant way our embedding of the *Circus* action type and communications.

**HOL-TestGen** [BW12] is a test-generation system based on Isabelle/HOL. The prover is used for all sorts of symbolic computations, which are controlled by a number of hand-written tactics. In particular, HOL-TestGen decomposes the initial property to be checked — the *test specification* $TS$ — after applying a number of case-splitting rules into a CNF-like normal form. The clauses of this normal form — called *test cases* — are simplified; detected empty test-clauses are eliminated. The result is a decomposition of the input-output relation of a test-specification over a variable $SUT$ representing logically the "system under test". More precisely, HOL-TestGen generates a *test theorem* of the form:

$$\frac{C_1(a_1) \Rightarrow \phi(a_1, SUT\ a_1) \quad ... \quad C_n(a_n) \Rightarrow \phi(a_n, SUT\ a_n) \quad \text{THYP}(H_1 \wedge ... \wedge H_m)}{TS}$$

The test-theorem states: whenever $SUT$ passes the tests — *i.e.* for a given constraint conjunct $C_i(a_i)$ a concrete instance for $a_i$ can be found, and the result of $SUT$ for this instance passes the oracle test $\phi$ — and whenever the test-hypothesis $H_1$-$H_m$ like *uniformity* and *regularity* hold (see [GLG08]), the test specification $TS$ holds. (THYP is just a constant used to keep the test-hypothesis generated during the splitting process; see [BW12] for details). The process of finding a concrete ground-instance for $C_i(a_i)$ is called test-data selection and uses various constraint-solvers, among them Z3 [DMB08].

While the test-generation method for *Circus* presented in this thesis is in spirit very similar to HOL-TestGen's, the systems are not connected (yet). While HOL-TestGen's case-spitting rules and control heuristics are geared towards the splitting of variables over data types such as lists and trees, the *Circus* procedure is based on splitting rules (see section 5.3.8) over *Circus* actions and configurations thereof. Its set of control heuristics is at present very different. The HOL-TestGen's test data

selection and instantiation can, nevertheless, be reused with little adaptations. This is possible essentially because of the generality of test data generation.

## 3.4 General Considerations

In order to clarify the current achievements limits of our work, we summarize here the testability, *i.e.* the assumptions on the SUT, and some restrictions that we made on the *Circus* language:

- The first testability hypothesis is a classical one, unavoidable in formal testing [HBH08]. It states that the SUT behaves like some unknown *Circus* process, with the same set of observations as the specification.

- The second testability hypothesis states that the SUT satisfies a complete testing assumption. This means that, after performing a test experiment a number of times on the SUT, all non-deterministic behaviors of this SUT are covered. This number of repetitions is different for each SUT and depends on its internal implementation. Such a property can be ensured by using adequate test drivers as in reachability testing [LC06] or in CHESS [BBC+10].

- Another hypothesis on the SUT is the presence of a reliable reset. It is essential, before starting a test experiment, to ensure that the SUT is in an initial state. The presence of a reset in SUT is not exactly a hypothesis, but a restriction on the systems that can be studied. This restriction is not a part of the testing theory, but it is important for the test execution process.

- The way of detecting deadlocks in our work can be problematic in divergent situations. Actually, divergence cannot be distinguished and is detected as a deadlock. Consequently, *Circus* specifications that describe a divergent behavior are not considered for test generation. This restriction has already been used in the literature *e.g.* TGV [JJ04].

- In general, all HOL types can be used as types for *Circus* variables and channels including generic types. This is allowed for reasoning about specifications (*e.g.* refinement proofs). However, only concrete HOL types are considered for test generation. This restriction is needed in particular for test instantiation, where values of generic types cannot be instantiated.

- The *Circus* language allows the description of complex communications, in addition to the simple ones. Complex communications are performed using a channel that can exchange different values at the same time. In our theories we consider only one way communications with an input or an output value.

However, the theories can be easily extended to cover complex communications. This is due to our representation of channels as functions (cf section 4.3).

- *Circus* specifications may contain all sorts of schema operations. In our theories, we consider only normalized schema operations as defined in the Z standard. This makes schema operations expressible as relations on the state variables. This is not a restriction since arbitrary Z schema can be normalized. Complex schema declarations are also not considered.

- In our theories, we will consider *Circus* processes as entities that cannot be composed. The operations over processes are not expressible at the moment. We believe, however, that compound processes can be written (by hand) as a process. The states of both processes will be then merged in one state, and the main actions of the processes will be composed in one main action. An automatic rewriting will be considered as a future extension of our theories.

## 3.5 Conclusions

This chapter presents a condensed introduction to the context of the thesis: *Circus* and Isabelle/HOL. *Circus* is introduced by giving (a big part of) its syntax for processes and actions, a simple example is used to show the overall process structure. A denotational and an operational semantics are associated to this syntax to describe semantically *Circus* actions and processes. These semantics are based on a unified logical framework (UTP), which is also introduced in this chapter. Two important applications based on *Circus* and its semantics are presented. The first application is process refinement, combining action refinement based on the denotational semantics and data refinement based on simulation relations. The second application is a testing theory from *Circus* specifications based on the operational and the denotational semantics.

The second part of the context is Isabelle/HOL theorem prover, used as a formal framework for all our work. Important basic notions that are relevant to our work are presented including types, logical connectives, sets pairs and lists. Additional advanced constructs of Isabelle/HOL are also introduced, this concerns constant and type definitions, type classes and extensible records. Our work was largely inspired from three applications based on Isabelle/HOL. These applications (HOL-Z, HOL-CSP and HOL-TestGen) are also introduced in this chapter.

After this introduction to the context, the next two chapters present our main contributions. Chapter 4 introduces our embedding of UTP and *Circus* denotational semantics in Isabelle/HOL. The central idea of this embedding is the representation of alphabets by extensible records and channels by functions. A part of the refine-

ment theory of *Circus* is also introduced in the next chapter, with the proof of some refinement laws and a simple example of refinement proof. Chapter 5 presents our representation of the operational semantics of *Circus* and the testing theory based on it. Automatic test generation tactics are defined in the spirit of HOL-TestGen using the operational semantics rules and some other derived rules.

CHAPTER 4

# Isabelle/Circus

## Contents

## 4.1 Introduction

This chapter introduces the first contribution of the thesis: the Isabelle/*Circus* specification and verification environment. It is based on the Isabelle generic theorem prover. This environment allows for writing *Circus* specifications and proving theorems over them. It contains proof rules and tactic support that allows for proofs of refinement for *Circus* processes (involving both data and behavioral aspects).

*Circus* actions are defined in the Isabelle/*Circus* environment as semantic entities, defined over HOL types. Concretely, the type of *Circus* actions is given as a shallow embedding of some particular UTP relations. This allows *Circus* variables and channels to have arbitrary HOL types. Therefore, the Isabelle/*Circus* environment offers a representation of the denotational semantics of *Circus* in Isabelle/HOL. As seen in Chapter 3, this denotational semantics was originally defined [OCW07] in terms of the UTP (Unifying Theories of Programming) [HH98], a semantic framework based on an alphabetized relational calculus. For this reason, we first introduce a formalization of UTP in Isabelle/HOL.

The Isabelle/*Circus* environment is defined in three layers: (i) the first layer is the formalization of (a substantial part of) UTP theories [FGW10], (ii) the second layer contains the representation of the *Circus* denotational semantics [FGW11, FGW12] and (iii) the top most layer is a proof system for *Circus* containing rules for refinement and simulation [FGW11, FGW12].

These layers are presented in the following sections. First, the UTP embedding in Isabelle/HOL is introduced in section 4.2. Then section 4.3 describes our formalization of the *Circus* denotational semantics. Finally, we introduce our proof system for refinement in section 4.4. We illustrate, with an example, how the environment may be used for refinement proofs.

## 4.2 Representing UTP in HOL

Textbook UTP presentations reveal a particular syntactic flavor of certain language aspects, a feature inherited from the Z tradition. The UTP framework is centered around the concept of *alphabetized* predicates, relations, etc, which are noted $(\alpha P, P)$ where $\alpha P$ is intended to produce the *alphabet* of the predicate $P$ associated to a superset of the free variables in it. In prior works based on the *ProofPower* theorem prover [PP], providing a formal semantics theory for UTP in HOL [OCW06, ZC09b, ZC10], the authors observed the difficulty that "the name of a variable is used to refer both to the name itself and to its value". For instance, in the relation

$$(\{x, x'\}, x > 0 \land (x' = x + 1 \lor x' = x - 1)), \tag{4.1}$$

the left-most $x, x'$ indicates the names $x$ resp. $x'$, while the right-most $x, x'$ stand for their values.

The starting point of any embedding of UTP is the representation of the alphabet (variable names) and the state space (association of names to values). Another important consideration when representing an object-language in a meta-language is the embedding level. In the literature [BGG+92], two major embedding styles can be distinguished: *deep embedding* and *shallow embedding*. The choice of the embedding level influences drastically the representation of the alphabet, the state space and the UTP constructs.

### *Deep vs. Shallow* embedding

There exists two well-known solutions for formalizing the semantics of the UTP (or any other formalism) in the logics of a theorem prover. The formalization may be either a *deep embedding* or a *shallow embedding*. Some examples of deep embeddings are: JAVA [ON99], UTP [OCW06] and B [JD07]. Other examples of shallow embeddings are: TLA [Mer95], CSP [TW97] and Z [BRW03].

Oliveira et al. [OCW06] proposed an embedding of the UTP in the *ProofPower* theorem prover. They aimed at the proof of refinement laws. For that, the authors made the choice of using a deep embedding to prove meta-theorems. In a deep embedding, an explicit data type for abstract syntax and an explicit semantic interpretation function is defined that relates syntax and semantic domain. While rather natural and easy to follow, such a representation has a number of drawbacks, both conceptually as well as practically *w.r.t.* the goal of efficient deduction:

1. there are necessarily ad-hoc limitations of the cardinality of the semantic domain *VAL* (e.g. sets are limited to be *finite* in order to keep the recursive definition of the domain well-founded),

2. the alphabet uses an untyped presentation — there is no inherent link from names and their type, which must be established by additional explicit concepts adding a new layer of complexity, and

3. the reasoning over the explicit alphabet results in a large number of side-conditions ("provisos") hampering deduction drastically.

In contrast to this deep embedding approach we opt for a shallow embedding. The characterizing feature for the latter is the following: if we represent an object-language expression $E$ of type $T$ into the meta-language by some expression $E'$ of type $T'$, then the mapping is injective for both $E$ and $T$ (provided that $E$ was well-typed with $T$). In contrast, deep embeddings define a surjective map from object-language expressions to a data type $AST$ (abstract syntax tree) in the meta-language. Due to injective map on types, the types are implicit in a shallow

representation, and thus reference to them via provisos associated to rules is unnecessary. It means that type-inference is used to perform a part of the deduction task beforehand, once and for all, as part of a parsing process prior to deduction.

### State space representation

The representation of the state space is crucial for the efficiency of our embedding. This representation should, on one hand, be expressive enough to cover all the aspects present in the UTP (and *Circus*). On the other hand, it should be efficient and transparent enough (and preferably shallow) to avoid introducing additional problems (see below). Different existing solutions for state representation in Isabelle/HOL are discussed in [SW09]. The authors listed some general features of state spaces that must be present in any representation. These features are *lookup and update*, *typing*, *modularity* and *scalability*.

The lookup and update are the most important features of a state space. The value of a name is retrieved using the lookup. The mapping of a value to a name is modified with the update. In most cases, variables are represented in a typed way in state spaces. The representation of the state space should preserve its typing. The modularity of the state representation is also very important in some cases. in the UTP for example, when composing two relations each with a specific state, the resulting relation state is composed from these specific ones. Finally, the representation should be scalable to support efficiently very large state spaces.

The authors discuss some existing state space representations, considering the features introduced above. The discussed representations are the following:

*Functions.* The state is represented as a function from names to values. The lookup corresponds to a function application and the update to the function update. The main drawback of this representation is that all values must have the same type. The solution of this problem is to define a global type as a (labeled) union of the different possible types (as an abstract data type for example). This will disadvantage the modularity since different states may have different types and thus a different global type. An other inconvenient of this representation is that the type checking system cannot detect type errors (since all values have the same type). Explicit projections and type assertions must be added to ensure a correct representation of the lookup and update.

*Tuples.* The state can be represented by tuples. Variables are not identified by names but by their corresponding positions in the tuple. Lookup and update must be encoded explicitly for each concrete application. The typing problems of the first representation are avoided and variables can be associated to different HOL types. The type checking system is used to detect all ill-typed

expressions. The main problems with this representation are the lack of modularity and generality. Composing different states requires special treatment like variable renaming and some transformations on lookup and update functions. This representation do not scale very well since tuples will be split for every expression which makes the proofs unnecessarily very heavy.

*Records.* The definition of (extensible) records in Isabelle/HOL is inspired from the state representation by tuples. In section 3.3.2.4, we introduced the record type as a special Cartesian product of the fields types. The lookup and update functions are generated automatically by the system from the record definition. HOL typing is used for the fields (variables) and the type checking system can be reused. Generality and extensibility are improved comparing to the representation in tuples. Modularity is slightly improved thanks to the extension field and local naming of record fields. Scalability of this representation is better than for tuples since record splitting is not always performed. Many useful theorems are generated automatically with the record definition which makes proofs on records easier.

*Abstract types.* The state can be represented as an abstract type. The lookup functions are characterized axiomatically. Update functions cannot be defined directly, but can be expressed in term of relations. Modularity and generality are limited in this representation since all variables should be known in advance.

*Locales.* Isabelle locales allows for defining localized theory scopes. The scope defined by a locale may contain local constant definitions, fixed assumptions and theorems over them. Locales enable renaming, merging and addition of locally fixed constants. In order to represent the state space in a locale, two type variables are used for names and values types. Local assumptions are made on the names (distinctness for example). Injection and projection functions are defined locally for each concrete variable type. Lookup and update functions are defined using the injection and projection functions that correspond to the variable type. This representation enables modularity and extensibility.

In a shallow embedding, the most convenient state representations are records and locales. Both representations are based on some automatically generated injection and projection functions to represent lookup and update. In a representation with locales, variable names can be addressed directly and thus operations over them can be defined (*e.g.* renaming). Unfortunately, locales are not first-class objects in HOL and, consequently, cannot be addressed in proof terms. In records, variable names are not represented directly but faked somehow with the use of lookup functions. The state is represented by a record and can be addressed directly in proof

terms. This allows reasoning about the whole state space as an HOL object, independently from its specific content. For these reasons, we use a representation of the state space using records.

Consequently, a price we are ready to pay is that there may be *rules* (manipulating variable names) in textbook UTP, which must be implemented by a *rule scheme* in our representation. That is, there will be specific tactic support that implements a rule scheme, *e.g.* , by inserting appropriate coercions in a more general rule and applying the result in a specific context. This technique has been used for the Z schema calculus by Brucker et al. [BRW03].

The essential idea of our shallow embedding is the use of records to represent both variable names and values. Thus, the equation (4.1) will be represented by the $\lambda$-abstraction:

$$\lambda\, \sigma \bullet \sigma.x > 0 \wedge (\sigma.x' = \sigma.x + 1 \vee \sigma.x' = \sigma.x - 1) \tag{4.2}$$

which is of type $\langle x \rightsquigarrow \mathbb{Z}, x' \rightsquigarrow \mathbb{Z}, ...\rangle \Rightarrow$ bool, which is, in other words, a set of records in HOL. The reader familiar with SML-like record-pattern-match notation may also recognize expression (4.2) as equivalent to:

$$\lambda\{x, x', ...\} \bullet x > 0 \wedge (x' = x + 1 \vee x' = x - 1)$$

In record notation, the order of the names is insignificant (in contrast to, say, a representation by tuples). We use *extensible* records — the dots represent the possibility of their extensions, allowing to build up the UTP in an incremental way similar to Brucker and Wolff's approach [BW08].

In a shallow embedding, the injective representation function must not be a one-to-one translation of operator symbols; rather, it can introduce coercions in $E'$ on the basis of object-language types. For example, it can be necessary to coerce isomorphically a $\langle x \rightsquigarrow \mathbb{Z}, x' \rightsquigarrow \mathbb{Z}, ...\rangle$ *set* predicate to a $(\langle x \rightsquigarrow \mathbb{Z}, ...\rangle \times \langle x \rightsquigarrow \mathbb{Z}, ...\rangle)$ *set* relation in order to support the semantics of the UTP dash-notation $x'$ in terms of relational composition. Similar compiler techniques are necessary to add or remove fields in extensible records, for example when entering and leaving the scope of a local variable declaration.

Given this way of dealing with alphabets via extensible records, we introduce the most general features of UTP: the concepts of alphabetized predicates and alphabetized relations, designs and reactive processes. We represent alphabetized predicates by sets of extensible records, and the relations by sets of pairs of extensible records; for the latter, there is already the theory `Relation.thy` in the Isabelle library that provides a collection of derived rules for sequential relational composition `_ o _` or operators for least and greatest fixpoints (`lfp`, `gfp`).

In order to support a maximum of common UTP look-and-feel, we define an Isabelle interface for UTP. Thus, we implement on the SML level implementing

Isabelle a function that computes for a term denoting an alphabetized predicate (a `cterm` in Isabelle terminology) its associated alphabet and term, be it in the format of an alphabetized predicate or an alphabetized relation. This function is suitably integrated into the command language of Isabelle such that we can define and query on the Isabelle level:

```
define_pred sample "({x::int,x'::int, ...}, x = x' + 1)"
```

This statement will be expanded internally into definitional constructions of an alphabetized predicate. In more detail, the alphabetized predicate mechanism expands internally the `define_pred` command as follows:

```
record    sample_type = x::int, x'::int
definition sample "sample ≡λA::sample_type. x A = x' A + 1"
```

where the latter introduces by default a *constant* `sample` with the right type and a *theorem* `sample_def` containing the constant definition for `sample` as described in the definition command.

### 4.2.1 Predicates and relations

In the UTP, an *alphabetized predicate* is a pair (*alphabet*, *predicate*) where the free variables appearing in the predicate are all in the alphabet, e.g. $(\{x, y\}, x > y)$. As such, it is very similar to the concept of a *schema* in Z. In the Isabelle/*Circus* environment, we represent alphabetized predicates by sets of (extensible) records, e.g. `{A. x A > y A}` where `A` are the extensible records with the alphabet $\{x, y\}$ and satisfying $x > y$.

In the UTP, an *alphabetized relation* is an alphabetized predicate where the alphabet is composed of input (undecorated) and output (dashed) variables. In this case the predicate describes a relation between input and output variables; for example $(\{x, x', y, y'\}, x' = x + y)$, is represented in Isabelle/*Circus* by `{(A,A'). x A' = x A + y A}`, which is a set of pairs, thus a relation.

Standard predicate calculus operators are used to combine alphabetized predicates. The definition of these operators is very similar to the standard UTP one, with some additional constraints on the alphabets.

We introduce the abbreviation $\alpha$ `alphabet` as a syntactic marker to highlight types that we use for alphabetized elements; on this basis, alphabetized predicates as sets of records are defined as follows:

```
types α  alphabet = "α"
types α  predicate = "α alphabet ⇒bool"
```

The standard logical connectives on predicates are then introduced:

```
definition true::"α predicate" where "true = λA. True"
```

```
2
3  definition false::"α predicate" where "false = λA. False"
4
5  definition not::"α predicate ⇒αpredicate" ("¬ _")
6  where "¬ P = λ A. ¬ (P A)"
7
8  definition conj::"[α predicate, αpredicate] ⇒αpredicate"(infix "∧")
9  where "P ∧ Q = λA. (P A) ∧ (Q A)"
10
11 definition disj::"[α predicate, αpredicate] ⇒αpredicate" ("_ ∨_")
12 where "P ∨ Q = λA. (P A) ∨ (Q A)"
13
14 definition impl::"[α predicate, αpredicate] ⇒αpredicate" ("_ ⟶_")
15 where "P ⟶ Q = λA. (P A) ⟶ (Q A)"
```

Our typing requires that all arguments range over the *same* alphabet. This is a significant restriction compared to textbook UTP, where all alphabets are merged (by the union of the underlying sets), in the style of Z. Thus, there are implicit coercions between sub-expressions in the UTP alphabetized predicates that have to be made explicit by suitable coercion functions. The insertion of such coercion functions can be done automatically. Using an SML- computation of the alphabet of each sub-expression, a new alphabet (record) containing the two alphabets is introduced. The resulting expression is defined over this new alphabet. The generation of new alphabets is optimized if one alphabet is included in the other. The resulting expression is defined over the more general alphabet, with no need to introduce a new one. This technique has already been applied by Brucker et. al for the definition of HOL/Z [BRW03].

Now universal and existential quantifications on UTP predicates are introduced in terms of HOL quantifications:

```
1  definition ex::"'β ⇒('β ⇒'α predicate) ⇒'α predicate" ("∃ _ _")
2  where "∃ x P ≡λA. ∃x. (P x) A"
3
4  definition all::"'β ⇒('β ⇒'α predicate) ⇒'α predicate" ("∀_ _")
5  where "∀ x P ≡λA. ∀x. (P x) A"
```

It remains to define the UTP alphabetized relations type as a HOL relation over $in\alpha P$ and $out\alpha P$. Several programming constructs are also defined over relations. First, the *conditional statements* where the condition is represented as a predicate over $in\alpha P$, and the symbols are kept as defined in the UTP book.

```
1  types αrelation = "(α alphabet ×α alphabet) set"
2  types αcondition = "α ⇒bool"
```

```
3
4 definition
5 cond::"αrelation ⇒αcondition ⇒αrelation ⇒αrelation" ("_◁ _ ▷_")
6 where "(P ◁ b ▷ Q) = λ(A, A'). (b A ∧P (A, A')) ∨
7                                 (¬ (b A) ∧Q (A, A'))"
```

Second, we give the definition of the *sequential composition*; it uses a predefined HOL relation operator, which is the relation composition. This operator corresponds exactly to the definition of sequential composition of alphabetized relations.

```
1 definition comp::"α relation ⇒αrelation ⇒αrelation" ("_ ;; _")
2 where "(P ;; Q) = P o Q"
```

Note that the mid alphabets should match in order to make this definition well-typed. If mid alphabets are different, the type checker will detect a type error in the composition expression.
Next, we deal with assignments. Since the alphabet is defined as an extensible record, an update function is generated automatically for every field. For example, let `a` be a field in some record, then there is the function `r(|a := x|)`, or represented internally `_update_ name a x`. We use this internal representation to define by a syntactic paraphrasing the update relation family defined as `{(A,A'). A'= (|a := E A|)}`.
The syntactic transformation of the assignment to the update function is instrumented as follows:

```
1 syntax
2 "_Assign"     :: "[idt, α⇒ β] ⇒α relation"   ("_ :== _")
3 translations
4 "x :== E"      => "{(A,A'). A' = _update_ name x (E A)}"
```

The first command `syntax` introduces a syntactic constant without any formal definition. The `translations` transforms a syntactic constant to produce a formal term or expression. In this case, the `_Assign` syntactic constant is translated to an assign relation by adding `_update` to the name of the variable.
A last construct is the *Skip* relation, which keeps all the variable values as they were. We use an equality over $in\alpha P$ and $out\alpha P$ to represent this. By using the record type for the alphabet, this equality is considered as values equality.

```
1 definition skip_r::"α relation" ("Π_r")
2 where "Π_r = λ(A, A'). (A' = A)"
```

The notion of *refinement* is equivalent to the universal implication of predicates, it is defined using the universal closure used in the UTP.

```
1 definition closure::"α predicate ⇒bool" ("[_]")
2 where "[ P ] = ∀A. P A"
```

```
3
4 definition refinement::"[αpredicate,α predicate] ⇒bool" ("_⊑_")
5 where "P ⊑Q =  [ Q ⟶P ]"
```

### 4.2.2 Designs theory

in the UTP, in order to explicitly record the termination of a program, a subset of alphabetized relations is introduced. These relations are called *designs* and their alphabet should contain the special boolean observational variable `ok` (and `ok'`). It is used to record the start and termination of a program.

The *Designs* theory is centered around this concept of termination which is captured by the extra name `ok`. Thus, we consider alphabets that contain at least this variable. This fits well with our representation of alphabets by extensible records: any theorem that we prove once and for all in the *Designs* theory will hold in future theories, too.

#### The name `ok`.

In short, the definition proceeds straightforwardly:

```
1 define_pred alpha_d "({ok::bool, ...}, true) "
```

However, it is worthwhile to look at the internal definitions generated here:

```
1 record alpha_d     = ok::bool
2 types 'α alphabet_d = "'α alpha_d_scheme alphabet"
3 types 'α relation_d = "'α alphabet_d relation"
```

In this construction, we use the internal type synonym `alpha_d_scheme` that Isabelle introduces internally for the cartesian product format where $\delta$ captures the possible type extension.

Since the definition of alphabets and relations uses a polymorphic type, we declare a new alphabet and relation type by instantiating this type to an extensible `alpha_d`. All the expressions defined for the first, more general type, will be directly applicable to this new specific type.

#### Designs.

Designs are a subclass of relations than can be expressed in the form:

```
1                    (ok ∧P) ⟶ (ok' ∧Q)
```

which means that if a program starts with its precondition $P$ satisfied, it will finish and satisfy its post condition $Q$. The definition of designs uses the previous definitions of relations and expressions.

```
definition
design::"α relation_d ⇒αrelation_d ⇒αrelation_d" ("(_ ⊢_)")
where " (P ⊢ Q) ≡λ(A, A'). (ok A ∧P (A, A')) ⟶(ok A' ∧Q (A ,A'))"
```

As seen above, `ok` is an automatically generated function over the record type `alpha_d`; it returns the value of the field `ok`.

Once given the definition of designs, new definitions for *skip* are stated as follows:

```
definition skip_d::"α relation_d" ("Π_d")
where "Π_d ≡(true ⊢Π_r)"
```

Our definitions make it possible to lead some proofs using Isabelle/HOL.More details about proofs are given in Sect. 4.2.5.

Some healthiness conditions are defined to ensure that a design satisfies some well-formedness properties [CW06] (see table 4.2). Four healthiness conditions, $H1$ to $H4$, are defined to characterize designs.

### 4.2.3 Reactive processes

Following the way the UTP describes reactive processes, more observational variables are needed to record the interaction with the environment. Three observational variables are defined for this subset of relations: `wait`, `tr` and `ref`. The boolean variable `wait` records if the process is waiting for an interaction or has terminated. `tr` records the list (trace) of interactions the process has performed so far. The variable `ref` contains the set of interactions (events) the process may refuse to perform. These observational variables defines the basic alphabet of all reactive processes called "`alpha_rp`".

Proceeding like we did with `ok`, we extend the alphabet with the variables `wait`, `tr` and `ref`. The corresponding extended alphabet and the definition of reactive processes are given in our *Reactive Process* theory. This kind of alphabet is called a *reactive alphabet*.

#### The names `wait`, `tr` and `ref`.

The variable `wait` expresses whether a process has terminated or is waiting for an interaction with its environment. The variable `tr` records the trace of events (interactions) the process has already performed. The `ref` variable is an event set, that encodes the events (interactions) that the process may refuse to perform at this state.

The new alphabet is an extension of the alphabet of designs, using the same construct: extensible records. The traces are defined as polymorphic events lists, and the refusals as polymorphic events sets.

```
datatype α event    = ev α
```

```
types    'α trace   = "('α event) list"
types    'α refusals = "('α event) set"

define_pred alpha_rp
      "(alpha_d ∪{wait::bool,tr::'α trace,ref::'αrefusals,...},true)"
```

and we add the handy type abbreviation:

```
types ('α, 'δ) relation_rp = "('α, 'δ) alpha_rp_scheme relation"
```

Again, the $\delta$ is used to make the record-extensions explicit.

**Reactive Processes.**

Reactive processes are characterized by three healthiness conditions defined over `wait`, `tr` and `ref`. The first healthiness condition **R1** states that a reactive process cannot change the history of performed event.

$$\mathbf{R1}\ P = P \wedge (tr \leq tr')$$

This healthiness condition is encoded as a relation, it uses a function $\leq$ on traces, which is defined in our theory.

```
definition R1::"('α, 'δ) relation_rp"
where "R1 (P) ≡λ (A, A'). P (A, A') ∧(tr A ≤tr A')"
```

To express the second healthiness condition **R2**, we use the formulation proposed by Cavalcanti and Woodcock [CW06].

$$\mathbf{R2}\ (P(tr, tr')) = P(<>, tr' - tr)$$

It states that a process description should not rely on what took place before its activation, and should restrict only the new events to be recorded since the last observation. These are the events in $tr' - tr$.

```
definition R2::"('α, 'δ) relation_rp"
where "R2 (P) ≡λ (A, A'). P (A(|tr:=[]|),A'(|tr:= (tr A ' - tr A) |))"
```

The last healthiness condition for reactive processes, **R3**, states that a process should not start if invoked in a waiting state.

$$\mathbf{R3}\ (P) = \Pi \triangleleft wait \triangleright P$$

A definition is given to $\Pi$ (Skip process), and the healthiness condition is expressed as a conditional expression over predicates.

```
definition R3::"('α, 'δ) relation_rp"
where "R3 (P) ≡(Π_rea ◁(wait o fst) ▷P)"
```

We can now define a reactive process as a relation over a reactive alphabet that satisfies these three healthiness conditions. This condition can be expressed as a functional composition of the three conditions.

```
definition R::"(α, δ) relation_rp"
where "R ≡R3 o R2 o R1"
```

### 4.2.4 CSP processes

A CSP process is a UTP reactive process that satisfies two additional healthiness conditions called *CSP*1 and *CSP*2 (all well-formedness conditions are summarized in table 4.2). A process that satisfies *CSP*1 and *CSP*2 is said to be CSP healthy.

As for reactive processes, a theory *CSP Process* corresponds to the CSP processes healthiness conditions. in the UTP, a reactive process is a CSP process if it satisfies two additional healthiness conditions **CSP1** and **CSP2**.

```
definition CSP1::"(α, δ) relation_rp"
where "CSP1 (P) ≡ λ (A, A'). (P (A, A')) ∨(¬ ok A ∧tr A ≤tr A')"

definition J_csp::"(α, δ) relation_rp"
where "J_csp ≡λ(A, A'). ok A ⟶ok A' ∧tr A = tr A' ∧ref A = ref A'
                              ∧ wait A = wait A' ∧more A = more A'"

definition CSP2::"(α, δ) relation_rp"
where "CSP2 (P) ≡ P ;; J_csp"
```

CSP basic processes and operators can be encoded using their definitions as reactive designs. Isabelle was used to prove that these reactive designs are CSP healthy.

### 4.2.5 Proofs

As mentioned above, the theories contains also proofs for some theorems and lemmas. In the relations theory, 100 lemmas are proved using 250 lines of proof, and in the designs theory 26 lemmas are proved using 120 lines of proof. Since our definitions are close to the library definitions of Isabelle/HOL, we can exploit the power of the standard Isabelle proof procedures. For example, we consider the proof of the *true*−; *left zero* lemma. These are almost the same proof steps as those used in the textbook proof.

```
lemma t_comp_lz: "(true;;(P ⊢Q)) = true"
 apply (auto simp: expand_fun_eq design_def rel_comp_def_raw mem_def)
 apply (rule_tac x="b(|ok:=False|)" in exI)
  by (simp add: mem_def)
```

In the previous proof we first apply some simplifications using the operators definitions (eg. *design_def*), then we fix the `ok` value to false and finally some simplifications will finish the proof.

| |
|---|
| $H1$ : A design may not make any prediction on variable values until the program has started.<br>`P = λ(A,A'). ok A →P (A,A')` |
| $H2$ : A design may not require non-termination.<br>`P(A,A'⦇ok:=False⦈)→P(A,A'⦇ok:=True⦈)` |
| $H3$ : If the precondition of a design is satisfiable, its postcondition must be satisfiable too.<br>`P = P ;; Π` |
| $H4$ : Exclude miracle design.<br>`P ;; true = true` |
| $R1$ : The execution of a reactive process never undoes an event that has already been performed.<br>`P = P ∧ λ (A,A'). tr A ≤ tr A'` |
| $R2$ : The behaviour of a reactive process is oblivious to what has gone before.<br>`P = λ(A,A'). P(A⦇tr:=[]⦈,A'⦇tr:=(tr A' - tr A)⦈)` |
| $R3$ : Intermediate stable states do not progress.<br>`P = Πrea ◁ wait o fst ▷ P` |
| $CSP1$ : Extension of the trace is the only guarantee on divergence.<br>`P = P ∨ (λ (A,A'). ¬ ok A ∧ tr A ≤ tr A')` |
| $CSP2$ : A process may not require non-termination.<br>`P = P ;; J`<br>`J = λ(A,A'). (ok A →ok A')∧tr A' = tr A ∧wait A' = wait A`<br>`∧ ref A' = ref A ∧more A' = more A` |

where Π is the relational Skip, `o` is the HOL functional composition operator and `fst` returns the first element of a pair.

Table 4.2: UTP Healthiness conditions

## 4.3 Circus Denotational Semantics

Isabelle/*Circus* is based on a "shallow embedding" of the *Circus* denotational semantics in Isabelle/HOL, enabling state variables and channels in *Circus* to have arbitrary HOL types. Therefore, the entire handling of typing can be completely shifted to the (efficiently implemented) Isabelle type-checker and is therefore implicit in proofs. This drastically simplifies the definitions, proofs, and makes the reuse of standardized proof procedures possible. Compared to implementations based on

a "deep embedding" such as [ZC09a] this drastically improves the usability of the resulting proof environment.

Our representation brings particular technical challenges and contributions concerning some important notions about variables. The main challenge was to represent alphabets and bindings in a typed way that preserves the semantics and improves deduction. Thus, we decided to provide a representation of bindings without an explicit management of alphabets. However, the representation of some core concepts in the unifying theories of programming (UTP) and *Circus* constructs (variable scopes and renaming) needed some care. Therefore, we propose a (stack-based) solution that allows the coding of state variables scoping with no need for renaming. This solution is a contribution to the UTP theory, which does not allow nested variable scoping. Some other challenging and tricky definitions (*e.g.* channels and name sets) are explained in the sequel of this section.

| | | |
|---|---|---|
| Process | ::= | **circusprocess** Tpar* name = PParagraph* **where** Action |
| PParagraph | ::= | AlphabetP \| StateP \| ChannelP \| NamesetP \| ChansetP \| SchemaP |
| | \| | ActionP |
| AlphabetP | ::= | **alphabet** [ vardecl$^+$ ] |
| vardecl | ::= | name :: type |
| StateP | ::= | **state** [ vardecl$^+$ ] |
| ChannelP | ::= | **channel** [ chandecl$^+$ ] |
| chandecl | ::= | name \| name type |
| NamesetP | ::= | **nameset** name = [ name$^+$ ] |
| ChansetP | ::= | **chanset** name = [ name$^+$ ] |
| SchemaP | ::= | **schema** name = SchemaExpression |
| ActionP | ::= | **action** name = Action |
| Action | ::= | **Skip** \| **Stop** \| Action ; Action \| Action $\Box$ Action \| Action $\sqcap$ Action |
| | \| | Action \ chansetN \| var := expr \| guard & Action \| comm $\rightarrow$ Action |
| | \| | **Schema** name \| ActionName \| $\mu$ var $\bullet$ Action \| **var** var $\bullet$ Action |
| | \| | Action $[\![$ namesetN \| chansetN \| namesetN $]\!]$ Action |

Figure 4.1: Isabelle/*Circus* syntax

The Isabelle/*Circus* environment allows the representation of processes in a syntax that is close to the textbook presentations of *Circus* (see Fig. 4.1). Similar to other specification constructs in Isabelle/HOL, this syntax is "parsed away", *i.e.* compiled into an internal representation of the denotational semantics of *Circus*, which is a formalization in the form of a shallow embedding of the (essentially untyped) paper-and-pencil definitions by Oliveira et al. [OCW07], based on UTP. *Circus* actions are defined as CSP healthy reactive processes.

In the UTP representation of reactive processes given in section 4.2.3 the process type is generic. It contains two type parameters that represent the channel type

and the alphabet of the process. These parameters are very general, and they are instantiated for each specific process. This could be problematic when representing the *Circus* semantics, since some definitions rely directly on variables and channels (*e.g.* assignment and communication). In this section we present our solution to deal with this kind of problems, and our representation of the *Circus* actions and processes.

In the following, we describe the foundation as well as the semantic definition of the process operators of *Circus*. A distinguishing feature of *Circus* processes are explicit state variables which do not exist in other process algebras like, e.g., CSP. These can be:

- *global* state variables, *i.e.* they are declared via alphabetized predicates in the `state` section, or Z-like $\Delta$ operations on global states that generate alphabetized relations, or

- *local* state variables, *i.e.* they are result of the variable declaration statement **var** var • Action. The scope of local variables is restricted to Action.

On both kind of state variables, logical constraints may be expressed.

### 4.3.1 Circus variables

In order to cater for the set of variables in scope, the *Circus* semantics describes the alphabet of its components, be it on the level of alphabetized predicates, alphabetized relations or actions. We recall that these items are represented by sets of records or sets of pairs of records, following the idea that an alphabet is used to establish a "binding" of variables to values. The *alphabet of a process* is defined by extending the reactive process alphabet with the corresponding variable names and types. Considering the example *FIG*, where the global state variable *idS* is defined, this is reflected in Isabelle/Circus by the extension of the process alphabet by this variable, i.e. by the extension of the Isabelle/HOL record:

```
1 record α alpha = α alpha_rp +  idS :: ID set
```

This introduces the record type `alpha` that contains the observational variables of a reactive process, plus the variable `idS`. Note that our *Circus* semantic representation allows "built-in" bindings of alphabets in a typed way. Moreover, there is no restriction on the associated HOL type. However, the inconvenience of this representation is that variables cannot be introduced "on the fly"; they must be known statically *i.e.* at type inference time. Another consequence is that a "syntactic" operation such as variable renaming has to be expressed as a "semantic" operation that maps one record type into another.

#### 4.3.1.1 Updating and accessing global variables.

Since the alphabets are represented by HOL records, we need a certain infrastructure to access data in them and to update them. The Isabelle representation as records gives us already two functions "select" and "update". The select function returns the value of a given variable name, the update function updates the value of this variable. Since we may have different HOL types for different variables, a unique definition for select and update cannot be provided. There is an instance of these functions for each variable in the record. The name of the variable is used to distinguish the different instances: for the select function the name is used directly and for the update function the name is used as a prefix. For example, for a variable named "x" the names of the select and update functions are respectively `x` and `x_update`.

Since a variable is characterized essentially by these functions, we define a general type (synonym) called `var` which represents a variable as a pair of its select and update function (in the underlying state $\sigma$).

```
types (β, σ) var = "(σ ⇒ β) * ((β ⇒ β) ⇒ σ ⇒ σ)"
```

For a given alphabet (record) of type $\sigma$, the type `(β, σ) var` represents the type of the variables whose value type is $\beta$ in this alphabet. One can then extract the select and update functions from a given variable with the following functions:

```
definition select :: "(β, σ) var ⇒σ ⇒ β"
  where select f ≡ (fst f)

definition update :: "(β, σ) var ⇒ β ⇒ σ ⇒ σ"
  where update f v ≡ (snd f) (λ _ . v)
```

Finally, we introduce a function called `VAR` to implement a syntactic translation of a variable name to an entity of type `var`.

```
syntax "_VAR" :: "id ⇒(β, σ) var" ("VAR _")
translations VAR x => (x, _update_ name x)
```

Note that in this syntactic translation rule, `_update_ name x` stands for the concatenation of the string `_update_` with the content of the variable `x`; the resulting `_update_x` in this example is mapped to the field-update function of the extensible record `x_update` by a default mechanism. On this basis, the assignment notation can be written as usual:

```
syntax
  "_assign" :: "id ⇒(σ ⇒β) ⇒(α, σ) action" ("_ `:=` _")
translations
  "x `:=` E"   => "CONST ASSIGN (VAR x) E"
```

and mapped to the *semantics* of the program variable `(x,x_update)` together with the universal `ASSIGN` operator defined in Section 4.3.3.2.

#### 4.3.1.2 Updating and accessing local variables.

In *Circus*, local program variables can be introduced on the fly, and their scopes are explicitly defined, as can be seen in the *FIG* example. In textbook *Circus*, nested scopes are handled by variable renaming which is not possible in our representation due to the implicit representation of variable names. Instead, we represent local variables by global variables, using the `var` type defined above, where selection and update involve an explicit stack discipline. Each variable is mapped to a list of values, not to only one value (as in state variables). Entering the scope of a variable corresponds to adding a new value as the head of the corresponding values list. Leaving a variable scope corresponds to removing the head of the list. The select and update functions correspond to selecting and updating the head of the list.

Note that this encoding scheme does not require to make local variables lexically distinct from global variables; local variable instances are just distinguished from the global ones by the stack discipline. This results in a dynamic scoping mechanism which is required by the operational semantics.

### 4.3.2 Synchronization infrastructure

#### 4.3.2.1 Name sets.

An important notion, used in the definition of parallel *Circus* actions, is name sets. A name set is a set of variable names, which is a subset of the alphabet. This notion cannot be directly expressed in our representation since variable names are not explicitly represented. Its definition is a bit tricky and relies on the characterization of the variables in our representation. As for variables, name sets are defined by their functional characterization. Name sets are only used in the definition of the binding merge function $MSt$, that describes the merged state after a parallel execution:

$$\forall\, v \bullet\ (v \in ns1 \Rightarrow v' = (1.v)) \land (v \in ns2 \Rightarrow v' = (2.v)) \land (v \notin ns1 \cup ns2 \Rightarrow v' = v)$$

The disjoint name sets $ns1$ and $ns2$ are used to determine which variable values (extracted from local bindings of the parallel components) are used to update the global binding of the process. A name set can be functionally defined as a binding update function, that copies values from a local binding to the global one. For example, a name set $NS$ that only contains the variable $x$ can be defined as follows in Isabelle/*Circus*:

```
1 definition NS lb gb ≡ x_update (x lb) gb
```

where `lb` and `gb` stands for local and global bindings, `x` and `x_update` are the select and update functions of variable `x`. Then the merge function can be defined by composing the application of the name sets to the global binding.

#### 4.3.2.2 Channels.

Reactive processes interact with the environment via synchronizations and communications. A synchronization is an interaction via a channel without any exchange of data. A communication is a synchronization with data exchange. In order to reason about communications in the same way, a datatype `channels` is defined using the channels names as constructors. For instance, in:

```
datatype channels = chan1 | chan2 nat | chan3 bool
```

we declare three channels: `chan1` that synchronizes without data , `chan2` that communicates natural values and `chan3` that exchanges boolean values.

This definition makes it possible to reason globally about communications since they have the same type. A drawback is that the channels may not have the same type: in the above example the types of `chan1`, `chan2` and `chan3` are respectively of types `channels`, `nat` ⇒ `channels` and `bool` ⇒ `channels`. Now, in the definition of some *Circus* operators, we need to compare two channels. Unfortunately, different channels like for example `chan1` and `chan2` cannot be compared since they don't have the same type. A solution would be to compare `chan1` with (`chan2 v`). The types are equivalent in this case, but the problem remains because comparing (`chan2 0`) to (`chan2 1`) will state inequality just because the communicated values are not equal. We can of course define an inductive function over the datatype `channels` to compare channels, but this is only possible when all the channels are known *a priori*.

Thus, we add some constraint to the generic channels type: we require the `channels` type to implement a function `chan_eq` that tests the equality of two channels. Fortunately, Isabelle/HOL provides a construct for this kind of restriction: the type classes (sorts) seen in the section 3.3.2.3. We define a type class (interface) `chan_eq` that contains a signature of the `chan_eq` function.

```
class ev_eq =
  fixes ev_eq :: "α ⇒α ⇒ bool"
begin end
```

Concrete channels type must implement the interface (class) " `ev_eq`" that can be easily defined for this concrete type. Moreover, one can use this class to add some definition that depends on the channel equivalence function. For example, a trace (event list) equivalence function can be defined as follows:

```
fun tr_eq where
  tr_eq [] [] = True | tr_eq xs [] = False | tr_eq [] ys = False
| tr_eq (x#xs) (y#ys) = if ev_eq x y then tr_eq xs ys else False
```

This function will be applicable for traces of elements whose type belongs to the sort `ev_eq`.

### 4.3.3 Actions and processes

The *Circus* actions type is defined as the set of all the CSP healthy reactive processes. The type `(α,σ)relation_rp` is the reactive process type where $\alpha$ is of `channels` type and $\sigma$ is a record extensions of `action_rp`, *i.e.* the global state variables. On this basis, we can encode the concept of a process for a family of possible state instances. We introduce the vital type `action` via the type-definition:

```
typedef(Action)
 (α::ev_eq,σ) action = {p::(α,σ)relation_rp. is_CSP_process p}
  morphisms relation_of action_of
proof - {...} qed
```

As mentioned before, a type-definition introduces a new type associated to a (non-empty) set of possible values. In our case, this set is the set of reactive processes that satisfy the CSP-processes healthiness-conditions.

Technically, from this construct the system introduces two constants definitions `action_of` of type `"(α, σ) relation_rp ⇒ (α,σ) action"` and `relation_of` of type `"(α, σ) action ⇒(α,σ)relation_rp"` as well as the usual two axioms expressing their bijection:

1. `action_of (relation_of (X))= X` and

2. `is_CSP_process p ⟹ relation_of ( action_of (p))= p`

where the predicate `is_CSP_process` captures the healthiness conditions.

Every *Circus* action is an abstraction of an alphabetized predicate. Below, we introduce the definitions of all the actions and operators using their denotational semantics. We must provide for each action, the proof that this predicate is CSP healthy. In this section we show all *Circus* basic actions and operators definitions. We also show how a whole *Circus* process is represented in the UTP framework. The environment contains the definitions of all the *Circus* operators shown in the next section.

Moreover, Isabelle/*Circus* contains a proof of a theorem stating that every reactive design — based on the above and the subsequent definitions — is CSP healthy.

#### 4.3.3.1 Basic actions.

`Stop` is defined as a reactive design, with a precondition `true` and a postcondition stating that the system deadlocks and the traces are not evolving.

```
definition
Stop ≡   action_of (R (true ⊢λ(A, A'). tr A' = tr A ∧ wait A'))
```

Skip is defined as a reactive design, with a precondition *true* and a postcondition stating that the system terminates and all the state variables are not changed. We represent this fact by stating that the more field is not changed, since this field is mapped to all the state variables. Recall that the more-field is a result of our encoding of alphabets by extensible records and stands for all future extensions of the alphabet (e.g. state variables).

```
definition Skip ≡  action_of (R (true ⊢ λ (A, A'). tr A' = tr A
                                    ∧ ¬ wait A' ∧ more A = more A'))
```

#### 4.3.3.2 The universal assignment action.

In the previous section 4.3.1.1, we described how global and local variables were represented by access- and updates functions introduced by fields in extensible records. In these terms, the "lifting" to the assignment action in *Circus* processes is straightforward:

```
definition
  ASSIGN::"(β, σ) var ⇒(σ ⇒ β) ⇒(α::ev_eq, σ) action"
where
  ASSIGN x e ≡  action_of (R (true ⊢ Y))
where
 Y = λ(A, A'). tr A' = tr A ∧ ¬ wait A' ∧
                 more A' = (assign x (e (more A))) (more A)
```

where assign is the projection into the update operation of a semantic variable described in section 4.3.1.1.

#### 4.3.3.3 Internal and External Choice.

For the internal choice operator the semantics is quite simple. It is defined as the relational disjunction of the two actions.

```
definition ndet (infixl "⊓") where
P ⊓Q ≡   action_of ((relation_of P ) ∨(relation_of Q))
```

The external choice semantics is more complicated, the choice of one action or the other is made only in stable states. A stable state is characterized in the condition by $tr' = tr \wedge wait'$, this means that the actions are performing internal steps. One the actions are in a stable state *i.e.* they are ready to interact, the choice can be performed.

The external choice operator is defined in Isabelle/*Circus* as follows:

```
definition det (infixl "□") where
P □Q ≡   action_of(R (X ⊢ Y))
```

```
where
X = ¬(Spec F F (relation_of P)) ∧¬ (Spec F F (relation_of Q))
and
Y = (Spec T F (relation_of P)) ∧ (Spec T F (relation_of Q))
           ◁ λ (A, A'). tr A = tr A' ∧ wait A' ▷
     (Spec T F (relation_of P)) ∨ (Spec T F (relation_of Q))
```

where the operation *Spec* is defined as follows:

```
definition Spec x y P = λ(A, A'). P (A(|wait := y|), A'(|ok := x|))
```

#### 4.3.3.4 Guarded Actions.

A guarded action is an action that can be executed if the guard value is *true* only, otherwise it stops. A guard is defined as a predicate over the action variables and the semantics of the guarded action is defined as follows:

```
definition guard ("_ & _") where
g & P ≡  action_of(R (X ⊢ Y))
where
X = (g o more o fst) ⟶¬ (Spec F F (relation_of P))
and
Y = ((g o more o fst) ∧ (Spec T F (relation_of P))) ∨
    (¬ (g o more o fst) ∧ (λ (A, A'). tr A' = tr A ∧ wait A'))
```

Given this definition, it is easy to prove that if the value of the guard is *false* the action will stop. The proof of this property is given by the following:

```
lemma "false & P = Stop"
by (simp add: Guard_def Stop_def csp_defs design_defs utp_defs
                                                          rp_defs)
```

#### 4.3.3.5 Sequencing.

The sequence operator is defined using the UTP sequential composition operator: the semantics of composing two actions is given by the relational composition of their corresponding relations. No restriction is made on the alphabets of the relations, since it was already made in the definition of the relation composition operator.

```
definition seq (infixl ";") where
P ; Q ≡ action_of (relation_of P ;; relation_of Q)
```

#### 4.3.3.6 Schema Expressions.

In order to define the semantics of schema expressions, the function `Pre` is introduced. This function verifies that the precondition of a schema expression is *true* before applying the schema operation. This function is defined by existentially quantifying the output alphabet of the schema, since the precondition depends only on the input variables.

```
definition Pre ::"'α relation ⇒'α predicate" where
Pre sc ≡ λ A. ∃ A'. sc (A, A')
```

The semantics of schema expressions is then:

```
definition
Schema sc ≡  action_of(R (X ⊢ Y))
where
X = λ(A, A'). (Pre sc) (more A)
and
Y = λ(A, A'). sc (more A, more A') ∧ ¬ wait A' ∧ tr A = tr A'
```

#### 4.3.3.7 Prefixed actions and communications.

The definition of prefixed actions is based on the definition of a special relation `do_I`. In the *Circus* denotational semantics, various forms of prefixing were defined. In our theory, we define one general form, and the other forms are defined as special cases.

```
definition do_I c x P ≡  X  ◁ wait o fst ▷  Y
where
X = (λ (A, A'). tr A = tr A' ∧ ((c ‘ P) ∩ ref A') = {})
and
Y = (λ (A, A'). hd ((tr A') - (tr A)) ∈ (c ‘ P) ∧
     (c (select x (more A))) = (last (tr A')))
```

where `c` is a channel constructor, `x` is a variable (of `var` type) and `P` is a predicate. The `do_I` relation gives the semantics of an interaction: if the system is ready to interact, the trace is unchanged and the waiting channel is not refused. After performing the interaction, the new event in the trace corresponds to this interaction.

The semantics of the whole action is given by the following definition:

```
definition Prefix c x P S ≡ action_of(R (X ⊢ Y)) ; S
where
X = true
and
Y =  do_I c x P ∧ (λ (A, A'). more A' = more A)
```

where c is a channel constructor, x is a variable (of type var), P is a predicate and S is an action. This definition states that the prefixed action semantics is given by the interaction semantics (do_I) composed with the semantics of the continuation (the action S).

Different types of communication are considered:

- Inputs: the communication is done over a variable.
- Constrained Inputs: the input variable value is constrained with a predicate.
- Outputs: the communications exchanges only one value.
- Synchronizations: only the channel name is considered (no data).

The semantics of these different forms of communications is based on the general definition above.

```
definition read c x P ≡ Prefix c x true P
definition write1 c a P ≡ Prefix c (λs. a s, (λ x. λy. y)) true P
definition write0 c P ≡ Prefix (λ_.c) (λ_._, (λ x. λy. y)) true P
```

where read, write1 and write0 corresponds respectively to *input*, *output* and *synchronization*, *constrained input* corresponds to the general definition.

We configure the Isabelle syntax-engine such that it parses the usual communication primitives and gives the corresponding semantics:

```
translations
  c ? p →P      == CONST read c (VAR p) P
  c ? p : b →P  == CONST Prefix c (VAR p) b P
  c ! p →P      == CONST write1 c p P
  a → P         == CONST write0 (TYPE(_)) a P
```

#### 4.3.3.8 Hiding.

The hiding operator is interesting because it depends on a channel set. This operator P \ cs is used to encapsulate the events that are in the channel set cs. These events become no longer visible from the environment. The semantics of the hiding operator is given by the following reactive process:

```
definition
Hide ::"[(α, σ) action , α set] ⇒(α, σ) action" (infixl "\")
where
P \ cs ≡ action_of( R(λ (A, A').
          ∃ s. (relation_of P)(A, A'(|tr :=s, ref := (ref A') ∪ cs|))
              ∧ (tr A' - tr A) = (tr_filter (s - tr A) cs))); Skip
```

The definition uses a filtering function `tr_filter` that removes from a trace the events whose channels belong to a given set. The definition of this function is based on the function `ev_eq` we defined in the class `ev_eq` in Section 4.3.2. This explains the presence of the constraint on the type of the action channels in the hiding definition, and in the definition of the filtering function below:

```
fun tr_filter::"a::ev_eq list ⇒a set ⇒a list" where
  tr_filter [] cs = []
| tr_filter (x#xs) cs = (if (¬ chan-in_set x cs)
                           then (x#(tr_filter xs cs))
                             else (tr_filter xs cs))
```

where the `chan-in_set` function checks if a given channel belongs to a channel set using `ev_eq` as equality function.

#### 4.3.3.9 Parallel Composition.

The parallel composition of actions is one of the most important definitions in our environment. It involves two important notions presented in the last section, namely name sets and channel sets. As explained in Section 4.3.2, name sets are used in the state merge function *MSt* that merges the final values of the local states into the global one. The name sets are defined as state update functions that can be composed to build the global state by the *MSt* function. On this basis, we define the *MSt* function in Isabelle/Circus as follows:

```
definition
MSt s1 s2 = (λ (S, S'). (S' = s1 S)) ;; (λ (S, S'). (S' = s2 S))
```

where `s1` and `s2` are disjoint name sets.

The second important notion in this definition is channel set. As explained in section 4.3.2, channels are defined as datatype constructors. As channels are usually defined over different types, channel sets cannot be directly defined since the types of elements may be not the same (as explained in section 4.3.2). To avoid this problem, we use the communications type `channels` as the type of elements in the channel sets. Thus, in the case of constructors communicating values, we apply them to some dummy values to obtain a value of type `channels`. One can define, for instance, the channel set `cs = {chan1, chan2(Some x)}`, and define a new channels membership function over this channel set using the function `ev_eq`.

Given these definitions, the parallel composition operator is stated as follows:

```
definition Par ("_ ⟦_ | _ | _ ⟧_") where
A1 ⟦ ns1 | cs | ns2 ⟧ A2 ≡   action_of(R (X ⊢ Y))
where
X = (λ (S, S'). ¬∃ tr1 tr2. (Spec F F (relation_of A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F (wait S) (relation_of A2) ;;
```

```
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs) = (tr_filter tr2 cs)
 ∧ ¬ ∃ tr1 tr2. (Spec F (wait S) (relation_of A1) ;;
(λ (S, S'). tr1 = tr S)) (S, S') ∧(Spec F F (relation_of A2) ;;
(λ (S, S'). tr2 = tr S)) (S, S') ∧(tr_filter tr1 cs)=(tr_filter tr2 cs))
and
Y = (λ (S, S'). (∃ s1 s2. (λ (A, A'). (Spec T F (relation_of A1) (A, s1))
  ∧ Spec T F (relation_of A2) (A, s2)) ;; M_par s1 ns1 s2 ns2 cs) (S, S'))
```

where `A1` and `A1` are *Circus* actions, `ns1` and `ns2` are name sets defined as update functions over the state of the actions `A1` and `A2`. Finally, `cs` is a channel set defined over the communications type of the actions `A1` and `A2`.

Isabelle/*Circus* contains also the definitions of `tr_filter`, `M_par` and some other functions used in these definitions.

#### 4.3.3.10 Recursion.

To represent the recursion operator "$\mu$" over actions, we use the universal least fix-point operator "*lfp*" defined in the HOL library for lattices and we follow again [OCW07]. The most substantial deviation from the standard CSP denotational semantics (which requires Scott-domains and complete partial orderings), is that *Circus* actions form a complete lattice. Consequently, the least fix-points are used in [OCW07] to define recursive *Circus* actions. The operator *lfp* is inherited from the predefined "*Complete Lattice class*" under some conditions, and all theorems defined over this operator can be reused. In order to reuse this operator, we have to show that the least-fixpoint over monotonic actions operators, produces an `action` that satisfies the CSP healthiness conditions. This consistency proof for the recursion operator is the largest contained in the Isabelle/*Circus* library.

In order to reuse the *lfp* operator and its inherited proofs, we must prove that the *Circus* actions type defines a complete lattice. This requires us to prove that the actions type belongs to the "*Complete Lattice class*" of HOL. Since type classes in HOL are hierarchic, we provide a proof in three steps. First, we prove that the *Circus* actions type forms a lattice by instantiating the HOL "*Lattice class*". In the second step, we prove that actions type instantiates a subclass of lattices called "*Bounded Lattice class*". The last step is to prove the instantiation from the "*Complete Lattice class*". The details of these proofs are given in the theories documentation [FWG12].

```
instantiation action :: (ev_eq, type) lattice
begin
definition inf_action:
    inf P Q ≡ action_of ((relation_of P) ⊓ (relation_of Q))
definition sup_action:
    sup P Q ≡ action_of ((relation_of P) ⊔ (relation_of Q))
```

```
definition leq_action:
    P ≤ Q ≡ P ⊑ Q
definition less_action:
    P < Q ≡P ⊑ Q ∧ ¬ Q ⊑ P
instance proof
  {...}
end
```

A lattice is a partial order with infimum and supremum of any two actions, the $\sqcap$ (meet) and $\sqcup$ (join) operations select such infimum and supremum actions. The instantiation proof of the lattice class requires the introduction of the definitions of the *meet*, the *join* and the ordering operators $\leq$ and $<$. In addition to the definition, the instantiation provides some proof obligations to ensure that all the notions are well defined (e.g. the ordering relation is reflexive and transitive). After proving these properties, the action type is considered as a lattice.

```
instantiation action :: (ev_eq, type) bounded_lattice
begin
definition bot_action:
    bot ≡  action_of true
definition top_action:
    top ≡  action_of (R(true ⊢false))
instance proof
  {...}
end
```

For the instantiation of the bounded lattice class, we add definitions of bounds (top and bottom of the lattice) and prove that these bounds are well defined w.r.t the ordering relation.

```
instantiation action :: (ev_eq, type) complete_lattice
begin
definition Sup_action:
    Sup S ≡ if S={} then bot else action_of ⨆(relation_of ‘S)
definition Inf_action:
    Inf S ≡ if S={} then top else action_of ⨅(relation_of ‘S)
instance proof
  {...}
end
```

Finally, a complete lattice is a partial order with general (infinitary) infimum of any set of actions, a general supremum exists as well. The general $\bigsqcap$ (meet) and $\bigsqcup$ (join) operations select such infimum and supremum actions. These operations indeed determine bounds on this complete lattice structure. The Knaster-Tarski Theorem

(in its simplest formulation) states that any monotone function on a complete lattice has a least fixed-point. This is a consequence of the basic boundary properties of the complete lattice operations. Instantiating the complete lattice class allows one to inherit these properties with the definition of the least fixed-point for monotonic functions over *Circus* actions.

#### 4.3.3.11 Circus processes

A *Circus* process is defined in our environment as a local theory by introducing qualified names for all its components. This is very similar to the notion of *namespaces* popular in programming languages. Defining a *Circus* process locally allows us to encapsulate definitions of alphabet, channels, schema expressions and actions in the same namespace. It is important for the foundation of Isabelle/*Circus* to avoid the ambiguity between local process entities definitions. It is ensured by prefixing, e.g. `FIG.Out` and `DFIG.Out` in the example of section 4.4.

## 4.4 Using Isabelle/Circus

We describe the front-end interface of Isabelle/*Circus*. In order to support a maximum of common *Circus* syntactic look-and-feel, we have programmed at the SML level of Isabelle a compiler that parses and (partially) pretty prints *Circus* process given in the syntax presented in Figure 4.1.

### 4.4.1 Writing specifications

A specification is a sequence of paragraphs. Each paragraph may be a declaration of alphabet, state, channels, name sets, channel sets, schema expressions or actions. The main action is introduced by the keyword `where`. Below, we illustrate how to use the environment to write a *Circus* specification using the `FIG` process example presented in Figure 3.1.

```
1 circusprocess FIG =
2   alphabet = [v::nat, x::nat]
3   state = [idS::nat set]
4   channel = [req, ret nat, out nat]
5   schema Init = idS := {}
6   schema Out = ∃a. v' = a ∧ v' ∉ idS ∧ idS' = idS ∪{v'}
7   schema Remove = x ∈ idS ∧ idS' = idS - {x}
8   where var v • Schema Init; (μ X •(req →Schema Out; out!v →Skip)
9                                □ (ret?x →Schema Remove); X)
```

Each line of the specification is translated into its corresponding representation. In the following, we describe the result of executing each command:

- `circusprocess` command introduces a scope of local components whose names are qualified by the process name (`FIG` in the example).
- `alphabet` generates a list of record fields to represent the binding. These fields map names to value lists.
- `state` generates a list of record fields that corresponds to the state variables. The names are mapped to single values. This command, together with `alphabet` command, generates a record that represents all the variables (for the `FIG` example the command generates the record `FIG_alphabet`, that contains the fields `v` and `x` of type `nat list` and the field `idS` of type `nat set`).
- `channel` introduces a datatype of typed communication channels (for the `FIG` example the command generates the datatype `FIG_channels` that contains the constructors `req` without communicated value and `ret` and `out` that communicate natural values).
- `schema` allows the definition of schema expressions represented as an alphabetized relation over the process variables (in the example the schema expressions `FIG.Init`, `FIG.Out` and `FIG.Remove` are generated).
- `action` introduces definitions for *Circus* actions. These definitions are based on the denotational semantics of *Circus*. The type parameters of the action type are instantiated with the locally defined channels and alphabet types.
- `where` introduces the main action as in `action` command (in the example the main action is `FIG.FIG` of type `(FIG_channels, FIG_alphabet)action`).

### 4.4.2 Relational and functional refinement in Circus

The main goal of Isabelle/*Circus* is to provide a proof environment for *Circus* processes. The "shallow-embedding" of *Circus* and UTP in Isabelle/HOL offers the possibility to reuse proof procedures, infrastructure and theorem libraries already existing in Isabelle/HOL. Moreover, once a process specification is encoded and parsed in Isabelle/*Circus*, proofs of, eg, refinement properties can be developed using the ISAR language for structured proofs.

To show in more detail how to use Isabelle/*Circus*, we provide a small example of action-refinement proof. The refinement relation is defined as the universal reverse implication in the UTP. In *Circus*, it is defined as follows:

```
1 definition A1 ⊑c A2 ≡(relation_of A1) ⊑utp (relation_of A2)
```

where A1 and A2 are *Circus* actions, $\sqsubseteq$c and $\sqsubseteq$utp stand respectively for refinement relation on *Circus* actions and on UTP predicate.

This definition assumes that the actions A1 and A2 share the same alphabet (binding) and the same channels. In general, refinement involves an important data evolution and growth. The data refinement is defined in [SWC02, CSW03] by backwards and forwards simulations. Here, we give an example of a special case, the so-called *functional* backwards simulation. This refers to the fact that the abstraction relation R that relates concrete and abstract actions is just a function:

```
definition Simulation ("_ ≼_ _") where
 A1 ≼R A2 = ∀a b.(relation_of A2)(a,b) ⟶(relation_of A1)(R a,R b)
```

where A1 and A2 are *Circus* actions and R is a function mapping the corresponding A1 alphabet to the A2 alphabet.

### 4.4.3 Refinement proofs

We can use the definition of simulation to transform the proof of refinement into a simple proof of implication by unfolding the operators in terms of their underlying relational semantics. A problem with this approach is that the size of proofs will grow exponentially with respect to the size of the processes. To avoid this problem, some general refinement laws were defined in [CSW03] to deal with the refinement of *Circus* actions at operators level and not at UTP level. We introduced and proved this set of laws in Isabelle/*Circus* (see table 4.3).

In table 4.3, the relations "$x \sim_S y$" and "$g_1 \simeq_S g_2$" record the fact that the variable $x$ (repectively the guard $g_1$) is refined by the variable $y$ (repectively by the guard $g_2$) w.r.t the simulation function $S$.

These laws can be used in complex refinement proofs to simplify them at the *Circus* level. More rules can be defined and proved to deal with more complicated statements like combination of operators for example. Using these laws, and exploiting the advantages of a shallow embedding, the automated proof of refinement becomes surprisingly simple.

Coming back to our example, let us consider the DFIG specification below, where the management of the identifiers via the set idS is refined into a set of removed identifiers retidS and a number max, which is the rank of the last issued identifier.

```
circusprocess DFIG =
  alphabet = [w::nat, y::nat]
  state = [retidS::nat set, max::nat]
  channel = FIG.channels
```

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P;\ P' \preceq_S Q;\ Q'}\ \text{SeqI} \qquad \frac{P \preceq_S Q \qquad g_1 \simeq_S g_2}{g_1 \& P \preceq_S g_2 \& Q}\ \text{GrdI}$$

$$\frac{P \preceq_S Q \qquad x \sim_S y}{var\ x \bullet P \preceq_S var\ y \bullet Q}\ \text{VarI} \qquad \frac{P \preceq_S Q \qquad x \sim_S y}{c?x \to P \preceq_S c?y \to Q}\ \text{InpI}$$

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \sqcap P' \preceq_S Q \sqcap Q'}\ \text{NdetI} \qquad \frac{P \preceq_S Q \qquad x \sim_S y}{c!x \to P \preceq_S c!y \to Q}\ \text{OutI}$$

$$\frac{\begin{array}{c}[X \preceq_S Y] \\ \vdots \\ P\ X \preceq_S Q\ Y \qquad mono\ P \qquad mono\ Q\end{array}}{\mu\ X \bullet P\ X \preceq_S \mu\ Y \bullet Q\ Y}\ \text{MuI} \qquad \frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \Box P' \preceq_S Q \Box Q'}\ \text{DetI}$$

$$\frac{\begin{array}{cc}[Pre\ sc_1\ (S\ A)] & [Pre\ sc_1\ (S\ A) \qquad sc_2\ (A, A')] \\ \vdots & \vdots \\ Pre\ sc_2\ A & sc_1\ (S\ A, S\ A')\end{array}}{schema\ sc_1 \preceq_S schema\ sc_2}\ \text{SchI} \qquad \frac{P \preceq_S Q}{a \to P \preceq_S a \to Q}\ \text{SyncI}$$

$$\frac{P \preceq_S Q \quad P' \preceq_S Q' \quad ns_1 \sim_S ns_1' \quad ns_2 \sim_S ns_2'}{P [\![ns_1 \mid cs \mid ns_2]\!] P' \preceq_S Q [\![ns_1' \mid cs \mid ns_2']\!] Q'}\ \text{ParI} \qquad \frac{}{Skip \preceq_S Skip}\ \text{SkipI}$$

Table 4.3: Proved simulation laws

```
5   schema Init = retidS' = {} ∧max' = 0
6   schema Out = w' = max ∧ max' = max+1 ∧ retidS' = retidS - {max}
7   schema Remove = y < max ∧retidS' = retidS ∪ {y}  ∧ max' = max
8   where var w • Schema Init; (μ X •(req →Schema Out; out!w →Skip)
9                                □ (ret?y →Schema Remove); X)
```

We provide the proof of refinement of `FIG` by `DFIG` just instantiating the simulation function `R` by the following abstraction function, that maps the underlying concrete states to abstract states:

```
1 definition Sim A = FIG_alphabet.make (w A) (y A)
2                                 ({a. a < (max A) ∧ a ∉ (retidS A)})
```

where A is the alphabet of `DFIG`, and `FIG_alphabet.make` yields an alphabet of type `FIG_Alphabet` initializing the values of `v`, `x` and `idS` by their corresponding values from `DFIG_alphabet`: `w`, `y` and `{a. a < max ∧ a ∉ retidS}`).

To prove that DFIG is a refinement of FIG one must prove that the main action DFIG.DFIG simulates the main action FIG.FIG. The definition is then simplified, and the refinement laws are applied to simplify the proof goal. Thus, the full proof consists of a few lines in ISAR:

```
1 theorem "FIG.FIG ≼Sim DFIG.DFIG"
2   apply (auto simp: DFIG.DFIG_def FIG.FIG_def mono_Seq
3               intro!: VarI SeqI MuI DetI SyncI InpI OutI SkipI)
4   apply (simp_all add: SimRemove SimOut SimInit Sim_def)
5 done
```

First, the definitions of FIG.FIG and DFIG.DFIG are simplified and the defined refinement laws are used by the auto tactic as introduction rules. The second step replaces the definition of the simulation function and uses some proved lemmas to finish the proof. The three lemmas used in this proof: SimInit, SimOut and SimRemove give proofs of simulation for the schema Init, Out and Remove.

## 4.5 Conclusions

This chapter introduced our shallow embedding of the *Circus* language denotational semantics in Isabelle/HOL. This embedding is based on a formalization of the UTP in Isabelle/HOL. In particular, by representing the concept of UTP alphabet in form of extensible records, we achieved a fairly compact, typed presentation of the language. This shallow embedding allows arbitrary (higher-order) HOL-types for channels, events, and state-variables, such as, e.g., sets of relations etc. Besides, systematic renaming of local variables is avoided by compiling them essentially to global variables using a stack of variable instances. The necessary proofs for showing that the definitions are consistent *i.e.* satisfy is_CSP_healthy have been done, together with a number of algebraic simplification laws on *Circus* actions.

Since the encoding effort can be hidden behind the scene by flexible extension mechanisms of Isabelle, it is possible to have a compact notation for both specifications and proofs. Moreover, existing standard tactics of Isabelle such as auto, simp and metis can be reused since our *Circus* semantics is representationally close to HOL. Thus, we provide an environment that can cope with combined refinements concerning data and behavior. Finally, we show a small, but prototypic example of process specification and refinement proofs.

The whole formalization of UTP and *Circus* on top of it, forms the Isabelle/*Circus* environment. It can be used as a basis of a plenty of (formally defined) application. A first application was shown in this chapter, with the definition of a refinement notion for *Circus* that allows performing refinement proofs. The next chapter will introduce another substantial application based on the Isabelle/*Circus* environment.

This extension contains the basic notions of *Circus*-based testing with test generation techniques.

CHAPTER 5

# Semantics Based Testing

## Contents

## 5.1 Introduction

The present chapter explains in detail the second contribution of this thesis: *Circus*-based testing with Isabelle/HOL. The testing theory of *Circus* presented in Section 3.2.3.2 is based on its operational and denotational semantics. Isabelle/HOL makes it possible to define a testing approach based on the semantics of *Circus*. To distinguish this kind of approach from other, more syntax-based, approaches we call it *Semantics-based Testing*.

The general principles of our theorem prover-based test-generation approach are explained in Section 5.2. Section 5.3 introduces the encoding of the *Circus* operational semantics in Isabelle/HOL. This includes state representation and variable scoping, transition rules and some useful derived rules. Symbolic test definitions and generation tactics are explained in Section 5.4 and illustrated with a small example.

## 5.2 Theorem Prover-based test-generation

test-generation is subject of many research projects, and several approaches have been proposed. One important approach is the use of theorem provers for test-generation. The use of theorem provers is motivated by the fact that such kind of systems is verified and logically safe. Theorem provers offer a formal framework for any formal specification or verification method. In addition, test-generation systems can greatly benefit from the automatic and interactive proof techniques to define automatic and interactive test-generation techniques. Our test-generation system presented here belongs to this category of approaches. As mentioned in Section 3.3.3 our test-generation approach is developed in the spirit of HOL-TestGen.

As seen in Chapter 3 HOL-TestGen [BW12] is a test-generation system based on the Isabelle theorem prover. The main goal of this system is to use the facilities offered by Isabelle to help test-generation. First, a test specification is stated to describe the test set to be generated. Then different sorts of symbolic computations offered by the prover are used to transform and simplify this test specification. The result is an enumeration of test cases and some test selection hypotheses. For a test specification $TS$, the test theorem generated by the system can be stated as follows:

$$\frac{C_1(a_1) \Rightarrow \phi(a_1, SUT\ a_1) \quad ... \quad C_n(a_n) \Rightarrow \phi(a_n, SUT\ a_n) \quad \text{THYP}(H_1 \wedge ... \wedge H_m)}{TS}$$

which can be read: whenever $SUT$ passes the tests — *i.e.* for a given constraint $C_i(a_i)$ a concrete instance for $a_i$ can be found, and the result of $SUT$ for this

instance passes the test oracle $\phi$ — and whenever the test-hypotheses $H_1..H_m$ hold, the test specification $TS$ is satisfied.

Let us now consider our approach for *Circus* test-generation; similar ideas are then used in this case. A test specification is first stated, for example to generate tests from a specification $SP$, one test specification would be: $\forall\, t \in Tests(SP) \rightarrow prog(t)$ where $Tests$ is the test-generation method, $t$ is a test and $prog$ is a free variable used to collect test cases. A first simplification is performed by the system by unfolding the definition of $Tests$. This depends on the definition of the conformance relation and on the test strategy. Symbolic computations are applied using specific definitions and rules and the resulting cases are described in the form of inference rules. The premises describe the test conditions (constraints) and the conclusions contain the actual tests (stored by $prog$). The non-feasible cases are automatically removed, since they have a false premise: the system proves it and removes these cases. Symbolic computations are essentially applications of case splitting and introduction or elimination rules defined from the operational semantics.

The operational semantics rules play a very important role in the test-generation, since they are the basis of the symbolic computations. Test specifications are introduced as inference rules, where the premises describe the test conditions and the conclusion gives the test structure. Symbolic computations and simplifications are mainly preformed on the premises. This cannot be done using the introduction rules[1] used to define the operational semantics, but requires the definition of elimination rules[2]. These elimination rules are derived from the operational semantics and used in test-generation. In some more complex situations, the test specification addresses sets of transitions and not single transitions. In this case, a third kind of rules is derived to handle sets of evolutions and not only single evolutions. All these rules and definitions are explained in the following sections.

## 5.3 Circus Operational Semantics

The configurations of the transition system for the operational semantics of *Circus* actions are triples $(c \mid s \models A)$ where $c$ is a constraint (a UTP condition) over the symbolic variables in use, $s$ an assignment of values to all *Circus* variables in scope, and $A$ a *Circus* action. The transition rules over configurations have the form: $(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)$, where $e$ is a label *i.e.* a pair (channel $\times$ symbolic variable) or $\varepsilon$.

The transition relation is also defined in terms of UTP and *Circus* actions. The formalization of the operational semantics is realized on top of Isabelle/*Circus*. In

[1] An introduction rule is an inference rule that introduces a construction in the conclusion.
[2] An elimination rule is an inference rule that eliminates a construction from the premises.

order to introduce the transition relation for all *Circus* actions, configurations must be defined first. The representation of configurations including constraints, states, actions and labels is given below. The definition of the transition relation is then introduced based on these definitions. Additional rules –very useful in test-generation– are derived from the transition relation rules.

Before giving more details on the representation of the rules, an important and fundamental notion must be discussed first. The operational semantics and testing theory of *Circus* are based on symbolic executions and symbolic variables. The representation of this symbolic execution in our testing theory can be either deep or shallow. As for the encoding of UTP in Isabelle/HOL (cf section 4.2), we discuss these two levels of embedding and justify our choice.

### 5.3.1 Symbolic execution: *deep vs. shallow embedding*

The test definitions and generation techniques introduced in [CG11] are based on symbolic execution and variables. Symbolic variables are syntactic names that represent some value without any typing information. These variables are introduced to represent a set of values (or a single, loosely defined, value), possibly constrained by a predicate. An alphabet $a$ is associated to all symbolic definitions of the testing theory containing the symbolic variable names.

A deep symbolic representation requires the definition of these symbolic notions on top of Isabelle/HOL. This is rather hard to realize and may introduce some inconsistency to our theory. The main problem is that symbolic variables are not typed. Consequently, all type information recorded in *Circus* variables is lost at this stage. Moreover, constraints must also be syntactic entities, and thus, cannot be directly expressed using HOL predicate calculus. To avoid these problems, one can represent symbolic variables in a typed way like ordinary variables. However, the representation of UTP variables in Isabelle/*Circus* (namely, extensible records) does not allow dynamic variable introduction and renaming. This is because the record structure defines its type, and consequently cannot be dynamically changed due to static type checking. Thus symbolic variables cannot be represented as UTP variables.

As an alternative to this deep symbolic execution, we opt for a shallow embedding. This embedding is based directly on the Isabelle symbolic representation and computation facilities. Isabelle, as a formal framework, provides powerful symbolic computation facilities that can be reused directly in our environment. This requires symbolic variables to be HOL variables, which are fresh semantic typed entities introduced by the prover. Expressions over these variables are written directly using HOL predefined operators or logical connectors. Consequently, constraints are also semantic entities represented as HOL predicates. With our representation, all the symbolic execution is carried out by Isabelle's symbolic computations.

This representation choice is more natural and wise since symbolic computations and higher-order manipulations are not of the same nature. It is clear that these two notions are of two different abstraction levels. This is not the case in deep symbolic execution which is represented in the same level of *Circus* definitions. In the shallow representation, low-level symbolic computations are the basis of high-level formal definitions.

This choice of embedding strongly influences the definition and representation of the operational semantics and testing theory. The impact of this choice is explained in the different sections. A lot of explicit symbolic manipulations (*e.g.* fresh symbolic variable introduction) are managed implicitly by the prover.

In the context of this chapter, we use the terms "symbolic execution" to refer to the explicit symbolic manipulations defined in [CG11]; we use the terms "symbolic computations" to refer to the Isabelle's symbolic manipulations used for the shallow symbolic execution.

### 5.3.2 Constraints

In the *Circus* testing theory [CG11], symbolic execution is used to define symbolically the transition relation of the operational semantics. The symbolic execution system used in this case is based on UTP constructs. Symbolic variables (values) are represented by UTP variables with fresh names generated on the fly. The (semantics of the) constraint is represented by a UTP predicate over the values of these symbolic variables.

The representation of the constraint in our testing environment depends on our representation of symbolic variables and our implementation of symbolic execution. We made the choice to represent symbolic variables directly as HOL variables. This choice of representation is justified in section 5.3.1.

Constraints are represented by HOL predicates over these symbolic (HOL) variables. In transition rules that can update the constraint (*e.g.* schema expressions or guarded actions), a fresh HOL variable will be introduced to represent the corresponding value in the constraint. Expressions of the form "(s; v = e)" that are used in some of the rules of operational semantics cannot be expressed directly. A semantically equivalent expression is provided according to the representation of the state; this point is discussed in detail in section 5.3.5.

### 5.3.3 Actions

The action component of the operational semantics defined in [CG11] is a syntactic characterization of some *Circus* actions. This corresponds to the syntax of actions defined in the denotational semantics. Special actions are defined to cover some particular situations *i.e.* local choice and local parallel blocks.

In our representation of the operational semantics, the action component is a semantic characterization of *Circus* actions. Section 4.3.3 introduces a definition of *Circus* action type as the set of reactive CSP processes (CSP-healthy reactive designs). Isabelle/*Circus* theories contain also definitions for basic *Circus* actions (*e.g.* Skip) and *Circus* operators over actions (*e.g.* ; for sequential composition). The *Circus* action type is given by `(Θ, σ) action` where Θ and $\sigma$ are polymorphic type parameters for *channels* and *alphabet*; these type parameters are instantiated for concrete processes.

Even if the denotational semantics of the actions is not directly used by the rule inference system, the type of actions in the configuration corresponds to the denotational type of actions. This representation of the operational semantics using the denotational semantics corroborate our claim that our testing method is semantics based. A consequence of this representation is that it makes it possible reasoning on both denotational and operational semantics. A very interesting exercise would be to prove, using Isabelle/*Circus* the soundness of the operational semantics *w.r.t.* the denotational one [WCGF07]. However, the required amount of work prevented us from tackling it during this thesis.

The denotational and operational semantics of the special actions are introduced. These actions are used in rules (5.17 to 5.23) for local choice blocks (⊞) and in rules (5.28 to 5.34) for local parallel blocks.

### 5.3.4 Labels

All the transitions over configurations are decorated with labels to keep a trace of the events that the system may perform. A label may refer to a communication with an input or output value, a synchronization (without communication) or an internal (silent) transition. In *Circus*, an input communication is represented by *chan*?*var* and an output communication by *chan*!*val* where *chan* is a channel, *var* is a variable and *val* is a value. A synchronization is represented by *chan* where *chan* is a channel. The labels keep track of these events as follows: *chan*?*symbvar* for inputs, *chan*!*symbvar* for outputs and *chan* for synchronizations where *symbvar* is a symbolic variable. Silent transitions are labeled by a special label $\varepsilon$, that corresponds to an internal evolution of the system.

As explained in section 4.3.2.2, the channels type is represented in our theory as a *datatype*; every channel is a function (constructor) that returns for a given communicated value an event of type *Channels*.

Transition Labels type is also defined as a *datatype* as follow:

```
1 datatype Θ TransitionLabel = Inp Θ| Out Θ| Sync Θ| ε
```

where Θ is a type parameter representing an event in our transition relation. The transitions are labeled by one of these labels, an event (*chan*, *symbvar*) is yielded

by applying the function `chan(symbvar)`. Labels are built either with $\varepsilon$ or by using one of the constructors. For example, the label $chan!symbvar$ is represented by `Out(chan(symbvar))` in the transition relation.

### 5.3.5 State

In the theory of *Circus* testing [CG11], the symbolic state is represented by a UTP condition over output (dashed) variables. Concretely, it consists of an assignment of symbolic values to all *Circus* variables in scope. Scoping is handled by variable introduction and removal and nested scopes are avoided using variable renaming. The state is updated in the operational semantics rules by composing one of the three relations: $var\, x$, $end\, x$ or $x := e$, that corresponds to variable scope delimiters and update of mapped symbolic values. The state is also used to build the constraints in expressions of the form "$s;\ v = expr$" that corresponds to an evaluation of the expression $expr$ in the context (binding) of the state $s$.

Our representation of the state is explained in the following, including variable introduction, removal and nested scopes representations. Subsequently $\{\!|\}$ denotes the empty state.

#### The representation

The state conceptually maps *Circus* variable names to symbolic values. It also manages local scopes by a standard compilation mechanism as a stack. Each variable binding in the state consists of a vector of values, which keeps track of nested statements. The representation of the state must preserve the properties of the transition relation as defined in the operational semantics.

The symbolic state is the binding of *Circus* variables to symbolic values of the form (variable $\mapsto$ symbolic variable). As explained in section 5.3.2, symbolic variables are represented by HOL variables. As a consequence, the symbolic state can be represented as a symbolic binding (variable name $\mapsto$ HOL variable). Following the representation of bindings by extensible records (cf section 4.2), the state corresponds to a record that maps field names to HOL variables: the state can be updated by usual record update functions; the evaluation of expressions (in constraints for example) is done using select functions over records. This representation is based on Isabelle's symbolic computations and Isabelle/HOL's extensible records definitions. Thus, it increases efficiency since a lot of superfluous simplifications are avoided to the system simplifier.

#### Variable scopes

We explained in section 4.2 that the main idea of representing alphabets as records raises the limitation that alphabets cannot be modified. Consequently, introducing,

removing or renaming variables cannot be expressed. To recover from this limitation, the representation of variables in the state must preserve the semantics of variable scopes and assignments in *Circus*.

Since the state is a mapping between variables and symbolic values, a nested scope of a variable should not affect this mapping. This means that the value of a global variable should be the same before *var* and after *end* of a local variable. One possible solution is to use a stack, in this case the state is no more a mapping (variable $\mapsto$ value) but rather (variable $\mapsto$ vector of values), which keeps track of nested statements. For example, for a *Circus* action containing $var\ x;\ x := 1;\ var\ x;\ x := 2$, the state must contain the mapping between $x$ and the vector [1,2]. The vector of values is represented in our theories by lists, the first element of the list is the value of the variable in the current scope, *var* and *end* correspond to adding and removing the first item of this list. The following example lists the different mappings variable/values in the state after each command.

| | |
|---|---|
| $var\ x$ | $x \Mapsto [?]$ |
| $var\ x\ ;\ x := 1$ | $x \Mapsto [1]$ |
| $var\ x\ ;\ x := 1;\ var\ x$ | $x \Mapsto [?, 1]$ |
| $var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2$ | $x \Mapsto [2, 1]$ |
| $var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2;\ end\ x$ | $x \Mapsto [1]$ |
| $var\ x\ ;\ x := 1;\ var\ x\ ;\ x := 2;\ end\ x\ ;\ end\ x$ | $x \Mapsto []$ |

This representation affects slightly the semantics of assignments and evaluations over the state in the operational semantics transition relation. The value of a variable is no more the corresponding value in the mapping but the first element of it. The *select* and *update* functions are modified to preserve their original semantics. For a variable $x$ a symbolic value $e$, the definitions of *var*, *end*, *select* and *update* are given as follows:

- $var\ x\ e \equiv x := e \# x$
- $end\ x \equiv x := tl\ x$
- $x\_update\ e \equiv (hd\ x) := e$
- $x\_select \equiv (hd\ x)$

where # is the list constructor, *tl* gives the tail of a list and *hd* gives the head of a list (cf section 3.3.1.3). Entering a scope is equivalent to adding a new value to the top of the list. This is done using *var*, which defines in this case the introduction of an initialized variable. Leaving a scope is defined by removing the first element of the list. Update and select functions affect only the top element of the list. These operators are only defined and used in the operational semantics; their semantics differ from the usual UTP operators, where nested scopes are not possible.

This modification will also affect the actions semantics because the state uses the same alphabet as the actions. The problem is not related to the alphabet itself, but to the bindings, since our representation does not make a difference between alphabets and bindings (names and values). Since the binding is now referring to a list of values, actions cannot anymore be defined using the same binding. To deal with this problem two solutions are possible. The first solution is use two different alphabets (bindings) for state and actions and introduce transformations between these two bindings. This solution will hamper efficiency by adding superfluous transformations.

We opted for a second solution which consists of changing (slightly) the use of variables in the denotational semantics of *Circus* actions that depends explicitly on variables. This means that global variables will be manipulated using classical select and update functions. Local variables in scopes will be manipulated using the new select and update functions defined above. This solution is simple and easy to implement and the modifications are transparent. It is also more convenient for an efficient and globally unified representation of the semantics.

### 5.3.6 Operational semantics rules

The operational semantics is defined by a set of inductive inference rules over the transition relation of the form:

$$\frac{C}{(c_0 \mid s_0 \models A_0) \xrightarrow{e} (c_1 \mid s_1 \models A_1)}$$

where $(c_0 \mid s_0 \models A_0)$ and $(c_1 \mid s_1 \models A_1)$ are configurations, $e$ is a label and $C$ is the applicability condition of the rule. Isabelle/HOL uses this specification to define the relation as least fixed-point on the lattice of powersets (according to Knaster-Tarski). From this definition the prover derives three kinds of rules:

- the introduction rules of the operational semantics used in the inductive definition of the transition relation,
- the inversion of the introduction rules expressed as a huge case-splitting rule covering all the cases, and
- an induction principle over the inductive definition of the transition relation.

The soundness of these rules *w.r.t.* the denotational semantics has to be proved: it is the subject of another on-going work in the *Circus* group [WCGF07]. In our work, we assume the soundness of these rules by stating some axioms.

The case-splitting rule can be instantiated to concrete patterns for the actions, say $A = Skip$ or $A = B \sqcap C$, and simplified using some knowledge from the denotational semantics. This results in a collection of *elimination rules*, which are essential for test-generation (as mentioned in section 5.2).

In the inductive definition of the transition relation, some rules make explicit introduction and use of arbitrary (symbolic) variables *e.g.* assignment or input communication. Since all HOL variables are typed, this explicit reference to external variables introduces additional type parameters to the inductive definition. This parametrized definition can be seen as a definition of families of operational semantics rules, depending on the instantiation of this type parameter. This is problematic, because it restricts all variables to be of the same type for any given *Circus* process in order to be applicable. In order to get over this restriction, two different solutions are possible.

The first possible solution is to use one general type for all the variables and channels. This general type gathers all the different types used in the process in a so-called tagged union type. This global type is used to instantiate the type parameter in the inductive definition of the rules. A projection function should be defined for each concrete type in order to retrieve it from the global type. Some simplification rules can be introduced to the simplifier in order to make type packing and unpacking more transparent. This solution has an important drawback which is the loss of type information when type checking *Circus* actions. The benefits of the representation of the state space as records are lost with this global type definition. The semantics of *Circus* actions that explicitly involve variables or events must be adapted to this representation. Thus, a second solution is proposed that solves the type problem by preserving type checking without changing the denotational semantics of *Circus* actions.

By analyzing the rules that contain the extra type parameter, two categories can be distinguished. The first category contains *Circus* actions that manipulate the state explicitly *e.g.* schema expressions and scoping. The second category concerns rules that make explicit use of communicated values *e.g.* input and output communications. The solution we adopt of the type parameter problem is to hide the expressions that have this parameter as type. This solution is explained in detail for the two categories of rules.

For the first category, two generic state update actions (*state_update_before* and *state_update_after*) are introduced in the denotational semantics. These actions provide a global scheme for all *Circus* actions that may update the state *i.e.* assignment, schema expressions and variable scoping (*var*, *let* and *end*). Using the denotational semantics as formalized in Isabelle/*Circus*, all these *Circus* actions that manipulate the state were proved to be special cases of one of these two generic actions. These theorems are given in Appendix C.1. As a consequence, generic rules can be defined in the operational semantics using these generic actions. No

explicit use of variables is done in these generic rules, only a state update relation is involved; thus no extra type parameter is needed. The specific transition rules for the other actions can be derived from these generic rules preserving the semantics of the original transition rules defined in [CG11].

We introduce in the following the denotational semantics of the generic state update actions:

```
definition
state_update_before::"'σ relation ⇒('ϑ,'σ) action ⇒('ϑ,'σ) action"
where
state_update_before sc Ac = action_of(R (X ⊢Y) ;; relation_of Ac)
where
X = λ(A, A'). (Pre sc) (more A)
and
Y  = λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A'
```

```
definition
state_update_after::"'σ relation ⇒('ϑ,'σ) action ⇒('ϑ,'σ) action"
where
state_update_after sc Ac = action_of(relation_of Ac ;; R (true ⊢Y))
where
Y = λ(A, A'). sc (more A, more A') & ¬wait A' & tr A = tr A'
```

The action (`state_update_before sc Ac`) updates the initial state using a state update relation `sc` then behaves like a given action `Ac`. Analogously, the action (`state_update_after sc Ac`) behaves first like the given action `Ac` then updates the intermediate state using the state update relation `sc`.

As seen above, all actions that update the state are proved to be particular cases of these global definition. An example is the schema expression action, that can be written using `state_update_before sc Ac` as stated by the following lemma:

```
lemma Schema-is_state_update_before:
            "Schema sc = state_update_before sc Skip"
<proof>
```

This states that a schema expression defined by a relation `sc` is equivalent to a state update action using the same relation `sc` followed by `Skip`.

The second category of rules that refer to the additional type parameter contains rules in which communicated values are used explicitly. For example, in a parallel synchronization of two output events happening on the same channel, the two exchanged values are stated to be equal in the constraint. Communicated values can be manipulated in three elements: transition labels, constraints and actions. For transition labels, the solution is to hide these values in the channels and use

directly an event instead of a channel and a value. Since channels are represented by functions from value types to events, this used event represents the application of the channel on this value. When present in constraints, the communicated value is always used in an equality: with a new symbolic variable or in parallel synchronizations. Instead of using a symbolic variable to represent the communicated value, a symbolic variable is used to represent the symbolic event. For parallel synchronizations, the equality can be reproduced by using equality on the events since channels are also assumed to be equal.

For communicated events used in actions (*i.e.* input and output prefixing), the solution is more tricky. The input communications introduce a new scope for the communication variable and initialize it with a symbolic value. The output communications send the symbolic value representing the evaluation of the communicated expression in the current state. The symbolic value and operations that manipulate it must be hidden in the definition of the actions. Two syntactical *envelopes* are defined to hide all these elements while preserving the original semantics of prefixed actions. These envelopes are defined as follows:

```
definition
iPrefix::"('σ⇒'ϑ)⇒'σ relation⇒(('ϑ,'σ) action⇒('ϑ,'σ) action)⇒
              'ϑ set ⇒('ϑ,'σ) action ⇒('ϑ,'σ) action"
where
iPrefix c i j S P ≡
   action_of(R(true ⊢(do_I c S) ∧(λ (A, A'). more A' = more A)))‘;‘ P
```

The first action `iPrefix c i j S P` represents an input prefixed action. It is defined over 5 parameters: `c` is a function that produces an event for a given state, `S` is a set of events and `P` is an action. The additional parameters `i` and `j` are not used in the semantics; they are used to hold additional information that will be used in the operational semantics.

```
definition
oPrefix::"('σ ⇒'ϑ) ⇒'ϑ set ⇒('ϑ, 'σ) action ⇒('ϑ, 'σ) action"
where
oPrefix c S P ≡
   action_of(R(true ⊢(do_I c S) ∧(λ (A, A'). more A' = more A)))‘;‘ P
```

The second action `oPrefix c S P` defines an output prefixed action with three parameters: a generalized event constructor, a set of events and ac action. All parameters are of known types, that depend only on the events type `'ϑ` or on the state type `'σ`. Using these actions in the operational semantics will avoid using additional type parameters in the rules.

The concrete definitions of input and output communication actions are defined using these envelopes. First, the universal input prefixed action is defined over a channel `c`, a variable `x` and an action `P` using `iPrefix`. The first parameter of

`iPrefix` which is the generalized event constructor is built by composing the channel (*i.e.* event constructor) with the variable selector over a state. The set of events is instantiated with the range of the channel function and the action is the same action. The two additional parameters are replaced by a state update relation (opening the scope of the variable) and a scoping action.

```
definition
read::"('v ⇒'ϑ) ⇒('v,'σ) var ⇒('ϑ, 'σ) action ⇒('ϑ, 'σ) action"
where
read c x P ≡
  iPrefix (λ A. c (select x A)) (λ (s, s'). ∃a. s' = increase x a s)
      (Let x) (range c) P
```

The restricted input prefixed action is also defined using `iPrefix` but with an additional parameter representing the set of valid input values. It is very similar to the universal input except for two differences. The set of events is represented by the image of the set of values under the channel function. The second difference is that the value of the variable in the new scope must be in the set of valid values.

```
definition
read1::"('v⇒'ϑ)⇒('v,'σ) var⇒'v set⇒('ϑ,'σ)action⇒('ϑ,'σ)action"
where
read1 c x S P ≡
  iPrefix (λ A. c (select x A)) (λ (s, s'). ∃a. s' = increase x a s
     ∧a∈S) (Let x) (c‘S) P
```

The output prefixed action is defined over a channel `c`, an expression over the state `a` and an action `P` using `oPrefix`. The generalized event constructor is built by composing the channel with the value of the expression in a given state. The set of events is instantiated with the range of the channel function and the action is the same action.

```
definition
write1::"('v ⇒'ϑ) ⇒('σ ⇒ 'v) ⇒('ϑ, 'σ) action ⇒('ϑ, 'σ) action"
where
write1 c a P ≡ oPrefix (λ A. c (a A)) (range c) P
```

These concrete definitions are based on an additional type `'v` that corresponds to the type of the communicated values. This type is used in the channel constructor functions, variable definitions, communicated expressions and value sets. All these elements are somehow hidden in the definition of `iPrefix` and `oPrefix` in a way that ensures to get rid of the additional type. In this situation, general rules can be introduced in the operational semantics based on `iPrefix` and `oPrefix`. Concrete rules can be then derived from these general rules to cover input and output communications. All these rules will be introduced in the next section.

### 5.3.7 Representing the introduction rules

In this section we consider all the rules of the operational semantics given in [CG11] and we present their corresponding representation in our theory. A list of all the operational semantics rules represented in our theory is given in Appendix C.2.1.

First, the generic rules defined to recover the extra type problem are introduced. These rules are not part of the original operational semantics, but are used to derive some important rules *e.g.* for assignment and schema expressions. As explained before, these rules are divided in two categories: state update rules and communication rules. The first category contains three rules and the second one contains two rules.

The first rule (5.1) of the first category covers the `state_update_before` action. This rules is applicable if the initial constraint is satisfiable and the state update relation is applicable to the current state. The output state $s'$ is introduced in the rule as a fresh HOL variable, so it can be seen as a fresh symbolic state. The effect of the rule is to apply internally the state update relation and continue the remaining action in one of the possible output states represented symbolically by $s'$.

$$\frac{c \qquad sc(s, s')}{(c \mid s \models \mathit{state\_update\_before}\ sc\ A) \xrightarrow{\varepsilon} (c \wedge sc(s, s') \mid s' \models A)} \tag{5.1}$$

The second rule (5.2) of the first category is the final step of the evolution of `state_update_after` where the action is `Skip`. The state update can be applied in this case following the same idea as for rule 5.1, producing a new symbolic state $s'$.

$$\frac{c \qquad sc(s, s')}{(c \mid s \models \mathit{state\_update\_after}\ sc\ \mathit{Skip}) \xrightarrow{\varepsilon} (c \wedge sc(s, s') \mid s' \models \mathit{Skip})} \tag{5.2}$$

The last rule (5.3) of the first category describes the evolution step of the `state_update_after` for general actions different from `Skip`. In this case, if the action can evolve independently from the state update to some new configuration, then the state update can be applied after this new configuration.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad A1 \neq \mathit{Skip}}{(c_1 \mid s_1 \models \mathit{state\_update\_after}\ sc\ A_1) \xrightarrow{l} (c_2 \mid s_2 \models \mathit{state\_update\_after}\ sc\ A_2)} \tag{5.3}$$

The second category of generic rules covers input and output communications. The first rule (5.4) of this category covers the general input prefixed action `iPrefix`. As said before in section 5.3.6, two additional parameter $i$ and $j$ are used in this action representing a state update relation and a scoping action. Applicability conditions are the same as for state update actions, and the target configuration

is defined using the new state and the scoping over the remaining action. The transition is labeled with an input symbolic event using the label constructor *in* and the symbolic event. This symbolic event is obtained by applying the channel function $d$ to the new symbolic state $s'$.

$$\frac{c \qquad i(s, s')}{(c \mid s \models iPrefix\ d\ i\ j\ S\ Ac) \xrightarrow{in\ (d\ s')} (c \mid s' \models j\ Ac)} \tag{5.4}$$

The last rule 5.5 of the second category covers the output prefixed action. This rule is simpler than the input action rule since it does not involve any additional parameters. The only applicability condition is the satisfiability of the initial constraint and the rule produces an output symbolic event label. The symbolic event is built by applying the event constructor function $d$ to the initial state. The resulting constraint and state are unchanged and the action corresponds exactly to the continuation action.

$$\frac{c}{(c \mid s \models oPrefix\ d\ S\ Ac) \xrightarrow{out\ (d\ s)} (c \mid s \models Ac)} \tag{5.5}$$

The general rules presented above cover special cases of actions that manipulate values of an extra type. The complete list of rules given in [CG11] is presented in the sequel, associated with its representation in our theories. Some rules are represented directly in the inductive definition of the transition relations and other are derived from the generic ones presented above. Each original rule is introduced (with a different numbering than [CG11]) and its representation is marked with (+).

The assignment rule (5.6) is the first example of state update rule derived from the general one 5.1. As said in section 5.3.6, using the denotational semantics, the assignment is proven to be equivalent to a state update relation where only one variable value is updated. Consequently, the rule will be similar to the state update rule where the variable value is updated with a new symbolic variable. This symbolic variable $w_0$ is introduced as a fresh HOL free variable and the symbolic output state is given by updating the initial state. The application of the expression $e$ to the current state $s$ replaces the variables in $e$ by their corresponding symbolic variables in this state.

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge (s;\ w_0 = e) \mid s;\ v := w_0 \models Skip)} \tag{5.6}$$

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge w_0 = e\ s \mid v\_update\ w_0\ s \models Skip)} \quad (+)$$

For schema expressions the rule (5.7) is also derived from the general state update rule 5.1. The schema is used as a state update relation defining a symbolic output state from the initial state. In the original rule, output variables occurring in the schema are substituted with fresh symbolic values in the current state. In our representation, no substitution is needed because the variables are already bound to fresh symbolic variables in the new symbolic state $s'$.

$$\frac{c \qquad (s;\ \text{pre}\ Op)}{(c \mid s \models Op) \xrightarrow{\varepsilon} (c \wedge (s;\ Op[w_0/v']) \mid s;\ v := w_0 \models Skip)} \quad v = out\alpha s \quad (5.7)$$

$$\frac{c \qquad Op(s, s')}{(c \mid s \models Op) \xrightarrow{\varepsilon} (c \wedge Op(s, s') \mid s' \models Skip)} \quad (+)$$

The second example of rules derived from state update rules considers variable scoping actions. First, the variable scope declaration rule (5.8) is derived from the general state update rule 5.1. The update relation in this case corresponds to the variable scope introduction in the state defined in section 5.3.5. All type checking and side conditions are not needed in the representation of the rules since all *Circus* variables are typed as well as HOL symbolic variables. The type checking system will ensure that both variables are of the same type.

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models \mathbf{var}\ x : T \bullet A) \xrightarrow{\varepsilon} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \boldsymbol{let}\ x \bullet A)} \quad (5.8)$$

$$\frac{c}{(c \mid s \models \mathbf{var}\ x \bullet A) \xrightarrow{\varepsilon} (c \mid var\ x\ w_0\ s \models \boldsymbol{let}\ x \bullet A)} \quad (+)$$

For scoped actions, the rules are defined from the `state_update_after` rules where the update relation corresponds to closing the variable scope. The *Skip* case rule (5.9) is derived from the general rule 5.2 where the *end* function is used to close the scope of the variable $x$ (see section 5.3.5).

$$\frac{c}{(c \mid s \models \boldsymbol{let}\ x \bullet Skip) \xrightarrow{\varepsilon} (c \mid s;\ \mathbf{end}\ x \models Skip)} \quad (5.9)$$

$$\frac{c}{(c \mid s \models \textbf{let}\ x \bullet Skip) \xrightarrow{\varepsilon} (c \mid end\ x\ s \models Skip)} \quad (+)$$

The step case of scoped actions is covered by rule 5.10 which is derived from the general one 5.3. The representation of this rule is not shown because it corresponds exactly to the original rule.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{let}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{let}\ x \bullet A_2)} \quad (5.10)$$

An other example of derived rules contains input and output communications whose rules are extracted from the general ones defined earlier. The output prefixed action rule is derived from 5.5 by replacing the corresponding definition of the action. A fresh symbolic HOL variable is introduced to represent the communicated value. The transition is labeled by the corresponding label $out\,(d\ w_0)$ as for the general `oPrefix` rule.

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{d!w_0} (c \wedge (s;\ w_0 = e) \mid s \models A)} \quad (5.11)$$

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{out\,(d\ w_0)} (c \wedge w_0 = e\ s \mid s \models A)} \quad (+)$$

The input prefixed action rules (5.12) are derived from the general rule for `iPrefix` 5.4. The original definition of input communications contains a predicate $T$ that restricts the type of the received values. In our representation of he semantics, the channels are well typed functions and the type checking system will ensure that received values are well-typed. The predicate can be omitted from the definition of the prefixed action and the rule is simpler. However, in the case of restricted input communications, this predicate is very important since it carries more information than just type information. This is a consequence of the difference between *Circus* types and HOL types. A second rule is introduced to cover the case of restricted input communications using a set of valid values $T$. The input communication rule introduces a variable to the state like for the variable introduction rule 5.8. The transition is labeled by $in\,(d\ w_0)$, where $w_0$ is a fresh symbolic HOL variable.

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models d?x : T \rightarrow A) \xrightarrow{d?w_0} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \textbf{\textit{let}}\ x \bullet A)} \tag{5.12}$$

$$\frac{c}{(c \mid s \models d?x \rightarrow A) \xrightarrow{in\,(d\,w_0)} (c \mid var\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

$$\frac{c \qquad w_0 \in T}{(c \mid s \models d?x \in T \rightarrow A) \xrightarrow{in\,(d\,w_0)} (c \wedge w_0 \in T \mid var\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

A special case of prefixed actions is synchronizations where no value is exchanged. The transition rule 5.13 describes the evolution of this action.

$$\frac{c}{(c \mid s \models d \rightarrow A) \xrightarrow{d} (c \mid s \models A)} \tag{5.13}$$

$$\frac{c}{(c \mid s \models d \rightarrow A) \xrightarrow{ev\,(d)} (c \mid s \models A)} \tag{+}$$

The sequence action rules are introduced directly in the inductive definition of the transition relations since they do not manipulate variables explicitly. The first rule for sequence (5.14) is the final step where the first action is *Skip*. The representation of this rule is omitted because it is the same as the original rule.

$$\frac{c}{(c \mid s \models \mathit{Skip}\,;A) \xrightarrow{\varepsilon} (c \mid s \models A)} \tag{5.14}$$

The second rule (5.15) for sequential composition contains the evolution step when the first action can evolve. The representation of the rule is almost the same, only a condition is added to the premises to ensure that the action is not *Skip*.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1\,;B) \xrightarrow{l} (c_2 \mid s_2 \models A_2\,;B)} \tag{5.15}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad A_1 \neq \mathit{Skip}}{(c_1 \mid s_1 \models A_1\,;B) \xrightarrow{l} (c_2 \mid s_2 \models A_2\,;B)} \tag{+}$$

The internal choice rules (5.16) describe an internal non-deterministic choice between two actions. The representation of the rules is the same as the original, it is then omitted here.

$$\frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_1)} \qquad \frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_2)} \tag{5.16}$$

In an external choice between two actions, the choice is controlled by the environment by interacting with the system. The choice is not made immediately, but only if one of he actions proposes a communication. A special action is used to define local blocks that ensure the evolutions of the two actions until a communication is proposed. This action is written $(\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)$.

The external choice rule (5.17) introduces a local block with the two actions in the initial constraint and state. The representation of this rule is not shown since it is not changed.

$$\frac{c}{(c \mid s \models A_1 \Box A_2) \xrightarrow{\varepsilon} (c \mid s \models (\mathsf{loc}\ c \mid s \bullet A_1) \boxplus (\mathsf{loc}\ c \mid s \bullet A_2))} \tag{5.17}$$

Local blocks allow actions to evolve until a communication is possible or to the end of their evolution (*Skip*). This behavior is described in three couples of dual rules. The first couple of rules (5.18 and 5.19) covers the final case where one of the actions is *Skip*. In this case the finished action is automatically chosen in the resulting configuration. The representation of these rules corresponds exactly to the rules themselves.

$$\frac{c_1}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet \mathit{Skip}) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A)) \xrightarrow{\varepsilon} (c_1 \mid s_1 \models \mathit{Skip})} \tag{5.18}$$

$$\frac{c_2}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet \mathit{Skip})) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models \mathit{Skip})} \tag{5.19}$$

The second couple of rules for local blocks (5.20 and 5.21) covers silent evolution of the local actions. If one of the actions can evolve silently to a given configuration, the whole local block will evolve following the same configuration.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{5.20}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{5.21}$$

The last couple of rules (5.22 and 5.23) represents the most important case where the choice is actually made. If one of the actions can interact with the environment using a communication, the whole block will follow the same evolution of this action.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \varepsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)} \tag{5.22}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \varepsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)} \tag{5.23}$$

Guarded actions define a conditional behavior using a guard $g$ defined as a predicate and an action $A$. The transition rule for guarded actions (5.24) requires as a premise that the guard predicate must be satisfiable in the current state. In this case, the guard predicate is added to the constraint, and the execution continues with the associated action. The expression $(g\ s)$ evaluates the guard $g$ in the current state $s$: it replaces the variables in $g$ with their corresponding symbolic variables.

$$\frac{c \wedge (s;\ g)}{(c \mid s \models g \,\&\, A) \xrightarrow{\varepsilon} (c \wedge (s;\ g) \mid s \models A)} \tag{5.24}$$

$$\frac{c \qquad (g\ s)}{(c \mid s \models g \,\&\, A) \xrightarrow{\varepsilon} (c \wedge g\ s \mid s \models A)} \tag{+}$$

The hiding operator is used to internalize some events whose channels are in a given channel set *cs*. The elementary case is when the action is *Skip*, the hiding will not affect this action since no interaction will be made. The first rule (5.25) describes this situation when hiding over *Skip* can only evolve silently to *Skip*. The representation of this rule is omitted because it is unchanged.

$$\frac{c}{(c \mid s \models Skip \setminus cs) \xrightarrow{\varepsilon} (c \mid s \models Skip)} \tag{5.25}$$

Hiding over other actions different from *Skip* requires the definition of additional functions. The first function is *chan* which is a partial function from labels to events defined as follows:

```
fun chan::"'ϑ label ⇒'ϑ option" where
    chan ε = None
  | chan (in e)  = Some e
  | chan (out e) = Some e
  | chan (ev e)  = Some e
```

This definition is different from the original one since it returns an event and not a channel because channels are not of the same type in our representation. This problem was already discussed in the denotational semantics 4.3.2.2 where a type class (`ev_eq`) defines a channel equality operator for events.

The second function needed in hiding rules is a channel set membership, that checks if the channel of an event is contained in the channels of a set of events. This function (`filter_chan_set`) is defined using `ev_eq` as follows:

```
definition filter_chan_set::"'ϑ ⇒'ϑ set ⇒bool" where
filter_chan_set a cs = ¬(∃ e∈cs. ev_eq a e)
```

The rules of hiding operator can be defined using these two functions. The first rule (5.26) covers the case of external event *i.e.* an interaction on a channel which is not in the hidden channel set. In this case the event is not hidden and the execution continues by hiding the remaining action. In the representation of this rule, the channel set *cs* is replaced with an event set *es* and the corresponding function is used to check channels membership.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l \neq \varepsilon \quad chan\ l \notin cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \setminus cs)} \tag{5.26}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l \neq \varepsilon \quad \mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ l))\ es}{(c_1 \mid s_1 \models A_1 \setminus es) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \setminus es)} \quad (+)$$

The last rule (5.27) covers the second case where the evolution of the action is internal. An internal evolution can either be with a silent transition or with an event that must be hidden, in both cases the transition will be labeled with $\varepsilon$. In the representation of this rule, event sets are used instead of channel sets and the corresponding channel membership function *filter_chan_set* is used.

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l = \varepsilon \vee \mathit{chan}\ l \in cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad (5.27)$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l = \varepsilon \vee \neg\ \mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ l))\ es}{(c_1 \mid s_1 \models A_1 \setminus es) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2 \setminus es)} \quad (+)$$

Like external choice, parallel composition requires a particular action to keep track of the evolution of parallel action blocks. This particular action is written $(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \llbracket\ cs\ \rrbracket (\mathsf{par}\ s_2 \mid x_2 \bullet A_2)$, where $x_1$ and $x_2$ are name sets and $cs$ is a channel set. This action is a bit different from the one defined in [CG11], where each block contains also a local constraint. The definition used in this thesis is given in [CG10] and uses only one global constraint. The local constraints are omitted because, in addition to be redundant with the global constraint, they prevent global restrictions over synchronization values.

The rule 5.28 describes a silent transition from the parallel operator to the parallel blocks operator. In the formalization of the rule, the channel set is represented by an event set and the name sets are represented by a special state update function (see section 4.3.2).

$$\frac{c}{(c \mid s \models A_1 \llbracket\ x_1 \mid cs \mid x_2\ \rrbracket A_2) \xrightarrow{\varepsilon} \left( \begin{array}{l} c \mid s \\ \models \\ \quad \left( \begin{array}{c} (\mathsf{par}\ s \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right)} \quad (5.28)$$

The parallel blocks may evolve in three different scenarios: (i) with final actions, (ii) with a non synchronized event or (iii) with a synchronization of the two blocks. The first case is described in rule 5.29, where the two actions cannot evolve anymore

(*i.e.* *Skip*). The parallel blocks are merged in one final configuration, where the action is *Skip* and the state is the merge of the local blocks states.

The representation of the rule depends on the representation of the name sets and the state merge operation $((\exists\, x_2' \bullet s_1) \land (\exists\, x_1' \bullet s_2))$. As explained in section 4.3.2, a name set is a special state update function, that copies some values from a local state to a global state. Using this definition, the state merge operation can be defined by the composition of the two name sets as follows:

```
definition
StateMerge::"'σ local_state ⇒'σ local_state ⇒'σ ⇒'σ"
where
StateMerge (s1, ns1) (s2, ns2) s = ns2 s2 (ns1 s1 s)
```

where the `local_state` is defined by a pair (state, name set) and the state merge will copy the corresponding values from local states to the global state.

$$\frac{c}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet \mathit{Skip}) \\ \lbrack\!\lbrack cs \rbrack\!\rbrack \\ (\mathsf{par}\ s_2 \mid x_2 \bullet \mathit{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid (\exists\, x_2' \bullet s_1) \land (\exists\, x_1' \bullet s_2) \\ \models \\ \quad \mathit{Skip} \end{array}\right)} \tag{5.29}$$

$$\frac{c}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet \mathit{Skip}) \\ \lbrack\!\lbrack cs \rbrack\!\rbrack \\ (\mathsf{par}\ s_2 \mid x_2 \bullet \mathit{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid \mathit{StateMerge}\ (s_1, x_1)\ (s_2, x_2)\ s \\ \models \\ \quad \mathit{Skip} \end{array}\right)} \tag{+}$$

The second possible evolution of the parallel block is when one of the actions evolves silently or with an event whose channel is not in the synchronization set. The full parallel block will evolve following the local evolution, without any synchronization. The rules 5.30 and 5.31 cover this situation. Only the representation of the first one is shown, the second is just the dual. Note that the channel membership function *filter_chan_set* is also used in this rule.

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \lor \mathit{chan}\ l \notin cs}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \lbrack\!\lbrack cs \rbrack\!\rbrack \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ \lbrack\!\lbrack cs \rbrack\!\rbrack \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{5.30}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee (\mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ l))\ es) \quad c_3}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \quad (+)$$

$$\frac{(c \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee \mathit{chan}\ l \notin cs}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_3 \mid x_2 \bullet A_3) \end{array}\right) \end{array}\right)} \quad (5.31)$$

The last –and more interesting– evolution is the synchronization of the two actions with events whose channels are in the synchronization set. In this situation, three different cases are possible depending on the resulting event type in the label. The first case is when the two actions can perform a synchronization on the same channel without exchanging values. This case is described in rule 5.32 where the resulting configuration is the synchronization of the two local configurations. The representation of this rule uses also the *filter_chan_set* as channel membership function for the synchronization event.

$$\frac{\begin{array}{c}(c \mid s_1 \models A_1) \xrightarrow{d} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{d} (c_4 \mid s_4 \models A_4) \\ d \in cs \quad c_3 \wedge c_4\end{array}}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d} \left(\begin{array}{l} c_3 \wedge c_4 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \quad (5.32)$$

$$\frac{\begin{array}{c}(c \mid s_1 \models A_1) \xrightarrow{ev\ d} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{ev\ d} (c_4 \mid s_4 \models A_4) \\ \neg\ (\mathit{filter\_chan\_set}\ d\ es) \quad c_3 \wedge c_4\end{array}}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{ev\ d} \left(\begin{array}{l} c_3 \wedge c_4 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \quad (+)$$

The second case is the synchronization of two input events on the same channel: this produces also an input event and the exchanged value is the same in both actions. This situation is described in rule 5.33 with its corresponding representation.

Since labels are represented by symbolic events, the distinction between the channel and the symbolic value is not possible. Fortunately, the equality of the two symbolic events implies the equality of channels and the equality of symbolic values.

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \quad d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d?w_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{5.33}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{in\ d_1} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{in\ d_2} (c_4 \mid s_4 \models A_4) \quad \neg\,(\mathit{filter\_chan\_set}\ d_1\ es) \quad \neg\,(\mathit{filter\_chan\_set}\ d_2\ es) \quad c_3 \wedge c_4 \wedge d_1 = d_2}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{in\ d_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge d_1 = d_2 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{+}$$

The last case of synchronization occurs when one action produces an output on some channel and the other action interacts on the same channel (with an input or an output). This case is described in rule 5.34 where the whole parallel block produces an output on the same channel. In the representation of this rule, event equality is used to represent channel and value equality and channel membership is done using *filter_chan_set*.

$$\frac{\left(\begin{array}{l} (c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \\ \qquad\qquad \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \\ \qquad\qquad \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \end{array}\right) \quad d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d!w_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{5.34}$$

$$\frac{\left(\begin{array}{c}(c \mid s_1 \models A_1) \xrightarrow{in\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{out\ d_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{out\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{in\ d_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{out\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge\ (c \mid s_2 \models A_2) \xrightarrow{out\ d_2} (c_4 \mid s_4 \models A_4)\end{array}\right) \quad \neg\,(filter\_chan\_set\ d_1\ es) \quad \neg\,(filter\_chan\_set\ d_2\ es) \quad c_3 \wedge c_4 \wedge d_1 = d_2}{\left(\begin{array}{l}c \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2)\end{array}\right)\end{array}\right) \xrightarrow{out\ d_2} \left(\begin{array}{l}c_3 \wedge c_4 \wedge d_1 = d_2 \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4)\end{array}\right)\end{array}\right)} \quad (+)$$

The last considered action is the recursion. Its rule defines an additional element in the configuration representing an environment for actions. This environment associates process variables to their corresponding actions. The behavior of recursion is described using two rules 5.35 and 5.36. The first rule introduces the mapping between the process variable and the action in the environment and continues the execution with this action. The second rule replaces the process variable with its corresponding action from the environment in order to continue with the recursion. The recursion may introduce nested scopes of the same variable, which is not supported in the original UTP scope definition. Some variable renaming is needed in order to avoid this problem.

The representation of this rule depends on our representation of recursion in Isabelle/*Circus*. As explained in section 4.3.3 the HOL least fixed-point operator *lfp* is used to define recursion over *Circus* actions. The rules of recursion are represented using only one rule that unfolds the *lfp* operator using a predefined HOL rule. This unfolding rule is only possible for monotonic functions over actions, this condition is proved to be satisfied for all considered *Circus* operators. No process environment is needed in this case and the variables renaming is avoided thanks to the state representation that supports nested scoping.

$$\frac{c}{(c \mid s \models \mu\ X \bullet A, \delta) \xrightarrow{\varepsilon} (c \mid s \models A, \delta \oplus \{X \mapsto A\})} \quad (5.35)$$

$$\frac{c}{(c \mid s \models X, \delta) \xrightarrow{\varepsilon} (c \mid s \models \delta\ X, \delta)} \quad (5.36)$$

$$\frac{monotonic\ A}{(c \mid s \models (\mu\ X \bullet A(X))) = (c \mid s \models A(\mu\ X \bullet A(X)))} \quad (+)$$

The transition relation is defined inductively using the rules presented earlier. The soundness of these rules *w.r.t.* the denotational semantics is still to be proved, but not in the context of our work. This requires the introduction of a denotational definition of the transition relation based on UTP. This definition is given in [WCGF07] where the soundness proof is already performed for an earlier version of the operational semantics of *Circus*.

### 5.3.8 Derived rules

As explained in section 5.2, operational semantics elimination rules are very important for test-generation. These rules are derived from the introduction rules of the transition relation. In addition to elimination rules, other rules are defined to handle sets of transition relations and not single transitions. These rules are used, for example, in the generation of a (possibly infinite) set of continuations. These two kinds of derived rules are explained in the sequel.

#### 5.3.8.1 Elimination rules

The inductive definition of the operational semantics states introduction rules of a labeled transition relation over configurations. Isabelle/HOL uses this specification to define the least fixed-point of the relation on the lattice of power sets (according to Knaster-Tarski). From that the prover derives the introduction rules, their inversion (as case-splitting rule), and an induction principle.

The case-splitting rule can be instantiated to concrete patterns for the actions, say $A = Skip$ or $A = B \sqcap C$, and simplified using some knowledge from the denotational semantics. This results in a collection of *elimination rules*, which are essential for test-generation. For each introduction rule, a corresponding elimination rule is derived. For example an introduction rule is given in the following:

$$\frac{c}{(c \mid s \models Skip \,;\, A) \xrightarrow{\varepsilon} (c \mid s \models A)}$$

The elimination rule corresponding to this rule is:

$$\frac{(c \mid s \models Skip \,;\, A) \xrightarrow{l} cf_1 \qquad \begin{array}{c}[cf_1 = (c \mid s \models A) \quad l = \varepsilon \quad c] \\ \vdots \\ Q\end{array}}{Q}$$

This rule can be used in a proof state where the premises contain a similar transition relation where the target configuration $cf_1$ is unknown. Using this rule, the unknown configuration is replaced with its value in the premise as well as the transition label.

The complete list of elimination rules of the operational semantics can be found in Appendix C.2.2.

#### 5.3.8.2 Additional derived rules

The definitions of the test sets in *Circus* requires us to reason about sets of events and not only single events. The operational semantics rules presented earlier are defined for single events and thus describe single transitions. In some cases, sets of transitions must be defined and thus rules over sets of events are needed. We derived a new set of equality rules based on the introduction and elimination rules of the operational semantics. An example is the equality rule for schema expressions given below:

$$\frac{}{\begin{array}{c}\{t \mid c_1\ s_1\ A_1\ l \bullet (c \mid s \models Op) \xrightarrow{l} (c_1 \mid s_1 \models A_1) \land P\ t\ c_1\ s_1\ A_1\ l\} \\ = \\ \bigcup s' \in \{s'' \mid Op(s, s'')\} \bullet \{t \mid c \land P\ t\ (c \land Op(s, s'))\ s'\ Skip\ \varepsilon\}\end{array}}$$

This rules describes all the possible evolutions of a schema expression $Op$ in a given state $s$ and with a given constraint $c$. This rule can be applied for any set of this form, were the free variable $P$ can be instantiated to any set definition predicate. The set elements $t$ are generic and depends on the definition of the set predicate $P$. In some cases, it will be used as an event or a label in order to compute the set of possible transition labels. In other cases, it will be used to describe the set of configurations that can occur after a given configuration.

The way of writing these rules is very important, especially the presented rule for schema expressions. All possible evolutions of the schema expression are given by a union over all the possible output states of the schema relation. Since schema expressions define state updates using relations, an infinite number of output states may be possible. This infinite set of output states is isolated using the global union operator. The remaining set definition is based on the predicate $P$ which addresses only one possible thus symbolic output state.

Similar rules are derived for each *Circus* action; the complete list of rules is given in Appendix C.2.3.

## 5.4 Symbolic test-generation with CirTA

*CirTA* stands for *Circus* Testing Automation, which is a test-generation environment

for *Circus* based on Isabelle/*Circus*. As explained in Section 3.2.3.2, testing for refinement in *Circus* is defined for two conformance relations: traces inclusion and deadlocks reduction. These conformance relations are based on the important notion of *cstraces*. In this section we give first the definition of *cstraces* – based on the *Circus* operational semantics – and how they are generated. The test-generation is then explained for each conformance relation, in addition to some useful definitions.

As explained in section 5.3.1, all symbolic definitions and computations are shallow. Consequently, all symbolic notions defined in [CG11] are symbolic from the Isabelle's point of view. In our higher-level point of view, all these notions are introduced and defined concretely.

### 5.4.1 *cstraces* generation

As informally explained in section 3.2.3.2, $cstraces(P)$ is the set of constrained symbolic traces of the process $P$. A *cstrace* is a list of symbolic events associated with a constraint defined as a predicate over the symbolic variables used in the symbolic events. Events are given by the labels (different from $\varepsilon$) of the operational semantics transitions. Some additional rules are defined in order to build those lists from the operational semantics rules. We introduce a relation noted "$\Longrightarrow$" and defined inductively by:

$$\frac{}{cf_1 \overset{[]}{\Longrightarrow} cf_1} \qquad \frac{cf_1 \overset{\varepsilon}{\longrightarrow} cf_2 \quad cf_2 \overset{st}{\Longrightarrow} cf_3}{cf_1 \overset{st}{\Longrightarrow} cf_3} \qquad \frac{cf_1 \overset{e}{\longrightarrow} cf_2 \quad cf_2 \overset{st}{\Longrightarrow} cf_3 \quad e \neq \varepsilon}{cf_1 \overset{e\#st}{\Longrightarrow} cf_3} \quad (*)$$

where $cf_1$, $cf_2$ and $cf_3$ are configurations.

The introduction rules (*) are represented in our theories in the same way as for the operational semantics rules (using an inductive relation definition). Some inverted (elimination) rules are also derived from this definition in the same way as for the operational semantics.

#### *cstraces* definition

The *cstraces* set definition is given in [CG11] using the trace composition relation (*) as follows:

**Definition 5.4.1.**

$$cstraces^a(c_0, s_0, P) = \{(st, \exists(\alpha c \setminus \alpha st) \bullet c) \mid s\ P_1 \bullet \alpha st \leq a \wedge (c_0 \mid s_0 \models P) \overset{st}{\Longrightarrow} (c \mid s \models P_1)\}$$

$$cstraces^a(\textbf{begin}\ state[x : T]P \bullet \textbf{end}) = cstraces^a(w_0 \in T, x := w_0, P)$$

One can read: the constrained symbolic traces of a given configuration are the constrained symbolic traces that can be reached using the operational semantics rules and starting from this configuration. The constrained symbolic traces of a given action are those of an initial configuration defined by a *true* constraint, an empty state and the this given action.

The shallow symbolic representation of this definition is simpler since the symbolic alphabet $a$ is not addressed explicitly. The symbolic constraint is also removed because it is described by the set predicate. The definition of *cstraces* is introduced in our theory as follows:

**Definition 5.4.2.**

$$cstraces(c_0, s_0, P) = \{st \mid s\ P_1 \bullet c_0 \wedge (c_0 \mid s_0 \models P) \stackrel{st}{\Longrightarrow} (c \mid s \models P_1)\}$$

Since the operational semantics rules contain premises that ensure the validity of the target constraint, the trace constraint is a part of the set predicate. In a higher point of view, the symbolic trace is represented by a set of concrete traces restricted by the final constraint.

### *cstraces* generation tactic

As explained in Section 5.2, tests are introduced as test specifications that will be used for test-generation. The same idea is applied for trace generation, where a proof goal is stated to define the traces a given system may perform. This statement is given by the following rule, for a given process $P$:

$$\frac{length(tr) \leq k \quad tr \in cstraces(P)}{Prog(tr)} \tag{5.37}$$

where $k$ is a constant used to bound the length of the generated traces.

Using proof techniques, a proof for this rule is started following a step by step trace generation tactic. In each step, different simplification rules are applied on the premises until no simplification is possible. The shallow symbolic definition of *cstraces* makes it possible to simplify the set membership operator to a predicate in the premises. This representation of the test specification as an inference rule is very elegant. The premises describe the generation goal and the conclusion stores the generation results. The derived elimination rules of the operational semantics are the basis of the computations performed on the premises.

The final step of the generation produces a list of propositions, describing the generated traces stored by the free variable *Prog*. This step is final since no more simplifications are possible on the undefined free variable *Prog*. The trace generation tactic is described by the following algorithm:

**Data**: k : the maximum length of traces

Simplify the test specification using the *cstraces* Definition 5.4.2;

**while** *length* $\leq k \wedge$ *more traces can be generated* **do**

Apply the elimination rules of (*) on the current goal;

Apply the elimination rules of the operational semantics on the resulting subgoals;

**end**

The test specification 5.37 is introduced as a proof goal in the proof configuration. The premise of this proof goal is first simplified using the definition of *cstraces* given in 5.4.2. Each constrained symbolic trace can be seen as a concrete trace at this stage of generation. The constraint is described in the premises by a predicate over the HOL variables contained in the trace. After the first step, the resulting proof goal premises contain predicates over the "$\Longrightarrow$" relation. The application of the elimination rules (*) on this proof goal generates three possible continuations: the empty trace, a trace preceded by an $\varepsilon$ and an event followed by a trace. Each possible continuation is described in a subgoal, describing a different possible trace.

The first resulting subgoal allows the construction of partial traces by considering empty continuations. The other two cases make it possible to construct the traces element by element using the operational semantics. The first element is given by the label of a transition relation and the "$\Longrightarrow$" relation is used to describe the continuation.

The elimination rules of the operational semantics are applied to the two last subgoals in order to instantiate the first element. The system will try to match the transition in the premise of the subgoal with the transitions of the operational semantics. If a matching is found, the transition is replaced by an instantiation of the configurations; this may lead to more than one subgoal if different matchings are possible (*e.g.* internal choice). Infeasible traces are described in subgoals whose premises are *false*. In this case, the system is able to close these subgoals automatically since *ex falso sequitur quodlibet*.

Specifications can describe an infinite recursive behavior and thus yields an infinite number of symbolic traces. The generation is then limited by a given trace length $k$, defined as a parameter for the whole generation process. The rules are applied repeatedly until this given depth $k$, or until there is no possible continuation. The list of subgoals corresponds to the list of all possible traces whose length do not exceed the given limit. The final proof state represents all the *cstraces* the system can perform. An example is given in section 5.7 to illustrate this trace generation process on a small specification.

The trace generation process is implemented in Isabelle as a tactic. The trace

generation tactic can be seen as an *inference engine* that operates with the derived elimination rules of the operational semantics and the trace composition relation. It is encoded in ML and added as a top-level tactic that can be used directly in Isabelle proofs. This tactic is given as follows:

```
1  fun trace_generation_tac ctxt k = SELECT_GOAL
2    let
3      fun rules i = ...;
4      fun os_rule_tac i = (CHANGED o FIRST) (rules i);
5       fun os_rules_tac = TRY o REPEAT_ALL_NEW (os_rule_tac);
6    in
7      (eres_Inst_tac ctxt [(("k", 0), string_of_Int k)] @{thm reg_hyp})
           1
8      THEN
9        REPEAT_ALL_NEW
10         ((empty_trace ctxt ORELSE' split_trace ctxt)
11              THEN_ALL_NEW os_rules_tac) 1
12     THEN distinct_subgoals_tac
13   end
```

The tactic is defined as a function with to parameter `ctxt` that represents the proof context and `k` which is the generation depth. Some predefined tactical combinators are used to describe the tactic (*e.g.* `FIRST`, `TRY` and `THEN`) Three local functions are defined to simplify the definition:

- `rules` (line 3) contains the operational semantics elimination rules.
- `os_rule_tac` (line 4) uses the first function to define an application attempt of the elimination rules.
- `os_rules_tac` (line 5) iterate -while possible- the application of the second function.

The main tactic is defined in three sequential steps composed using the tactical combinator `THEN`. The first step (line 7) applies a regularity hypothesis (see 5.5) to limit the length of generated traces to the value of `k`. The second step (lines 9-11) performs the trace generation repeatedly on all resulting subgoals using `REPEAT_ALL_NEW` combinator. The trace generation is done by applying one of the trace elimination rules `empty_trace` or `split_trace`. This corresponds to the application of the elimination rules of the relation "$\Longrightarrow$" defined in *. The symbolic events are retrieved by applying the operational semantics rules on all the resulting subgoals. The last step of the generation tactic is the application of a predefined tactic `distinct_subgoals_tac` to remove redundant subgoals (line 12).

This trace generation tactic needs to be installed as a proof method to the current theory. This is done using the `method_setup` that associates to a rule name an ML

expression that describes the tactic to apply. The definition of the proof method for trace generation is done as follows:

```
1 method_setup trace_generation =
2   Scan.lift Parse.nat --|
3     Method.sections Simplifier.simp_modifiers >>
4     (fn k => fn ctxt => SIMPLE_METHOD' (trace_generation_tac ctxt k))
```

The description of the method starts by parsing two parameters: a natural number and an optional simplifier modifier (lines 2-3). The first parameter is the length limit $k$ of the generated traces. The second optional parameter is used to add some simplification rules that will be used by the simplifier during the trace generation. After installing this new method, it can be used directly in proof scripts. For example, to generate traces of length less than 3, the defined proof method can be called as follows:

```
1 apply (trace_generation 3)
```

### 5.4.2 test-generation for traces refinement

The first studied conformance relation for *Circus*-based testing corresponds to the traces-inclusion refinement relation. This relation states that all the traces of the SUT belong to the traces set of the specification, or in other words, the SUT should not engage in traces that are not traces of the specification. Thus, a forbidden *cstrace* is defined by a prefix which is a valid *cstrace* of the specification followed by a forbidden symbolic event (continuation). As seen in Section 3.2.3.2 the set of forbidden continuations is called $\overline{csinitials}$, its definition is based on valid continuations given by the *csinitials* set. Because of the constrained symbolic nature of the *cstraces* and events, $\overline{csinitials}$ is not exactly the complement of *csinitials*.

#### *csinitials* definition

The *csinitials* set is the set of constrained symbolic events a system may perform after a given trace. The *csinitials* set is defined in [CG11] as follows:

**Definition 5.4.3.** *for every* $(st, c) \in cstraces^a(P)$ *we define*

$$csinitials^a(P, (st, c)) = \\ \{(se, c \wedge c_1) \mid (st@[se], c_1) \in cstraces^a(P) \wedge (\exists\, a \bullet c \wedge c_1)\}$$

Symbolic initials after a given symbolic trace are symbolic events that concatenated to this trace yield other valid symbolic traces. In this case, only events whose constraints are compatible with the trace constraint are considered.

The shallow symbolic representation of this definition is given as follows:

**Definition 5.4.4.** *for every* $tr \in cstraces(P)$ *we define*

$$csinitials(P, tr) = \{e \mid tr@[e] \in cstraces(P) \wedge (c \wedge c_1)\}$$

All explicit symbolic manipulation is removed, since they are implicitly handled by the prover. The constraint of the trace is not considered, since at this level $tr$ is considered as a single concrete value.

### *csinitials* generation tactic

The generation of *csinitials* is done using a similar tactic as for *cstraces*. Each resulting subgoal corresponds to one possible *csinitial* after a given *cstrace*. Unfortunately, this technique enumerates the elements of *csinitials* and does not generate the whole *csinitials* set, which is more important for test-generation. The reason is that the test specification is usually written in form of an implication and not as an equivalence. Consequently, the trace generation tactic based on case-splitting rules generates only a subset of *csinitials* because of the implication. In order to generate the full set of *csinitials*, a different test theorem is written using set equality. This test theorem is defined as follows:

$$\frac{S = csinitials(P, tr)}{Prog\ S} \tag{5.38}$$

in this case, the free variable *Prog* will record the set $S$ of all *csinitials* of $P$ after the (symbolic) trace $tr$.

Due to the form of the new test specification 5.38 for *csinitials* sets, the defined operational semantics rules cannot be used directly. The generic derived rules introduced in section 5.3.8.2 can be used to generate the set of possible events after a given configuration. Using these rules the *csinitials set* generation tactic becomes very simple. The rules are added as simplification rules to the system simplifier and the latter will apply them automatically when it is invoked.

An additional equality rule is defined for the trace-composition relation "$\Longrightarrow$", because the definition of *csinitials* is based on *cstraces*. This rule is similar to the derived rules introduced in section 5.3.8.2. Its definition is given by the following:

$$\frac{}{\begin{array}{c}\{t \mid \exists\, cf_1 \bullet cf \overset{tr}{\Longrightarrow} cf_1 \wedge P\ t\ cf_1\ tr\} = \\ \left(\begin{array}{l}\{t \mid tr = [] \wedge P\ t\ cf\ []\} \cup \\ \{t \mid \exists\, cf_1\ cf_2 \bullet cf \overset{\varepsilon}{\longrightarrow} cf_1 \wedge cf_1 \overset{tr}{\Longrightarrow} cf_2 \wedge P\ t\ cf_2\ tr\} \cup \\ \left\{\begin{array}{l} t \mid \exists\, cf_1\ cf_2\ e\ tr_1 \bullet cf \overset{e}{\longrightarrow} cf_1 \wedge cf_1 \overset{tr1}{\Longrightarrow} cf_2 \wedge \\ \qquad e \neq \varepsilon \wedge tr = e\#tr_1 \wedge P\ t\ cf_2\ (e\#tr_1) \end{array}\right\} \end{array}\right)\end{array}}$$

All sets defined from the trace-composition relation "$\Longrightarrow$" between two configurations $cf$ and $cf_1$ can be written as a union of three sets. The first one covers the case of empty traces, the configurations are then equal. The second set contains all continuations after an internal transition. The last set splits the transition to one simple transition with one element followed by a trace transition over the remaining tail of the trace.

### $\overline{csinitials}$ definition

The $\overline{csinitials}$ set contains the continuations that are not possible in the specification. In order to generate tests for the traces inclusion relation, we need to introduce the definition of $\overline{csinitials}$. This set contains the constrained symbolic events the system should refuse to perform after a given trace. These elements will be used to lead the SUT to execute a prohibited trace, an error is then detected if the SUT do so. The $\overline{csinitials}$ set is defined as follows:

**Definition 5.4.5.** *for every* $(st, c) \in cstraces^a(P)$ *we define*

$$\overline{csinitials}^a(P,(st,c)) = \left\{ (d.\alpha_0, c_1) \mid \left( \begin{array}{l} \alpha_0 = a(\#st+1) \wedge \\ c_1 = c \wedge \neg \bigvee\{c_2 \mid (d.\alpha_0, c_2) \in csinitials^a(P,(st,c))\} \end{array} \right) \right\}$$

The $\overline{csinitials}$ set is built from the $csinitials$ set: if an event is not in the accepted $csinitials$ it is added to the $\overline{csinitials}$ constrained with the constraint of the trace. If the event is in the $csinitials$ it will be added with the negation of its constraint.

The new symbolic variable $\alpha_0$ is defined as a fresh variable in the alphabet $a$ after the symbolic variables used in the symbolic trace $st$. The definition of this symbolic variable is part of the symbolic execution defined in [CG11] for *Circus* testing.

In our theories, the symbolic execution is carried out by the symbolic computations of the prover. Consequently, all explicit symbolic constructs are not added in the representation of $\overline{csinitials}$. This representation is introduced as follows:

**Definition 5.4.6.** *for every* $tr \in cstraces(P)$ *we define*

$$\overline{csinitials}(P,tr) = \{d.\alpha_0 \mid \neg\, Sup\{d.\alpha_0 \in csinitials(P,tr)\}\}$$

where the *Sup* operator is the supremum of the lattice of booleans which is predefined in the HOL library, *i.e.* generalized set union. No constraint is associated to the trace $tr$ because it is seen as a concrete trace at this level. Symbolic $\overline{csinitials}$ are represented by sets of events where the constraint is built using the set membership operator over the $csinitials$ set.

The test specification for the traces inclusion conformance relation is given by:

$$\frac{tr \in cstraces(P) \wedge e \in \overline{csinitials}(P, tr)}{Prog(tr@[e])} \tag{5.39}$$

The resulting symbolic tests are built from a (symbolic) trace of $P$ concatenated to an element of its $\overline{csinitials}$ set. The constraint is retrieved from the simplification of the set membership operator in the premises of the proof goal. Starting from the resulting symbolic sequences, the symbolic tests can be obtained by inserting special verdict events *pass*, *fail*, and *inc* as described in Section 3.2.3.2. This last step is not part of the general test-generation process, since it depends on the concrete implementation of the SUT.

### $\overline{csinitials}$ generation tactic

The generation of tests (smaller than a given length $k$) for traces inclusion is done in two stages. First, the trace generation tactic is invoked to generate the symbolic traces of length smaller than $k$. For each generated trace, the set of the possible *csinitials* after this trace is generated using the corresponding generation tactic. Using this set, the feasible $\overline{csinitials}$ are generated and added as a subgoal in the final generation state. The generation tactic can be represented in the following algorithm:

**Data**: k : the maximum length of tests

Generate *cstraces* using trace generation tactic for a length k;

**foreach** *generated trace tr* **do**

> Simplify the test specification (5.39) using the $\overline{csinitials}$ Definition 5.4.6;
>
> Generate the *csinitials* after *tr* using *csinitials set* generation tactic;
>
> Apply case-splitting and simplification rules to generate the elements of $\overline{csinitials}$;

**end**

The derived rules defined for sets of transitions are used in this case to generate the *csinitials set* after a given trace *tr*. The trace is seen as a concrete trace in a higher point of view, with a constraint which is expressed in the premises of the test specification. The elements of $\overline{csinitials}$ can be generated using this set and listed one by one using case-splitting rules. Each case describes a possible element of $\overline{csinitials}$, whose constraint is defined in the premises.

In order to introduce test verdicts into the generated tests, a recursive function is defined in the testing theory. Before introducing this function, the verdict data type must be defined first.

```
datatype 'Θ verdict_event = pass | fail | inc | int 'Θ
```

This defines a new type `verdict_event` with four constructors: the three verdicts `pass`, `fail` and `inc` and an interaction defined by `int`. The verdict introduction function is defined recursively on the trace and on a given *csinitials* event as follows:

```
fun add_verdict_trincl::"'Θ list ⇒'Θ ⇒'Θ verdict list" where
  add_verdict_trincl [] a = [pass, int a, fail]
| add_verdict_trincl (e#l) a = [inc, int e]@(add_verdict_trincl l a)
```

As seen in Chapter 3.2.3.2, a `pass` verdict is inserted after the trace, followed by the tested element and a `fail` verdict. All the elements of the trace are added preceded by an `inc` verdict.

### 5.4.3 test-generation for deadlocks reduction

The deadlocks reduction conformance relation, also known as *conf*, states that all the deadlocks must be specified. Testing this conformance relation aims at verifying that all specified deadlock-free situations should be accepted by the SUT. A deadlock-free situation is defined by a symbolic trace followed by the choice among a set of events the system should not refuse, *i.e.* if the system is waiting for an interaction after performing a valid trace, it should accept to perform at least one element of the proposed *csacceptances* set (see section 3.2.3.2).

From this definition, we note that the *csacceptances* set is potentially infinite. Indeed, adding elements to an acceptance set gives another acceptance set. For practical reasons, we consider a new definition, the $csacceptances_{min}$ set which is a set of minimal acceptance sets. Minimal means that removing any element from an acceptance set leads to a non-acceptance (possibly blocking) set.

In a *csacceptances* set, input and output events are not treated in the same way because they represent different situations. Inputs are controlled by the environment, consequently, any instance of the input symbolic value is accepted if the symbolic input event is accepted. Outputs are different since they are controlled by the system and the concrete output value may be chosen in a nondeterministic way. All instantiations of output symbolic events must be considered in order to cover this internal nondeterminism.

#### *csacceptances* definition

In order to distinguish input symbolic events from output symbolic events, a new set called *IOcsinitials* set is defined. This set contains, for a given configuration, the

constrained symbolic initials where input and output information is recorded. Since inputs and outputs are considered separately in the labels of the transition relation, the set of *IOcsinitials* is easy to define. It contains the set of labels (different from $\varepsilon$) of all possible transitions of a given configuration. The set of *IOcsinitials* is defined by the following:

**Definition 5.4.7.**

$$IOcsinitials^{a}_{st}(c_1, s_1, A_1) = \left\{ \begin{array}{l} (l, \exists(\alpha c_2 \setminus (\alpha(st@[l]))) \bullet c_2) \mid s_2,\ A_2 \bullet \\ \quad (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge l \neq \varepsilon \wedge \alpha(st@[l]) \leq a \end{array} \right\}$$

A symbolic acceptance set after a given trace must contain at least one symbolic event from each *IOcsinitials* set obtained from a stable configuration after this trace. In our representation of this definition the symbolic alphabets $a$ and $st$ are not addressed explicitly. In addition to this, the constraint is defined in the set predicate. The definition of *IOcsinitials* is given in our theories as follows:

**Definition 5.4.8.**

$$IOcsinitials(c_1, s_1, A_1) = \{l \mid c_2,\ s_2,\ A_2 \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge l \neq \varepsilon\}$$

The general definition of *csacceptances* was introduced in [CG11] as follows:

**Definition 5.4.9.** *for every* $(st, c) \in cstraces^{a}(P)$ *we define*

$$csacceptances^{a}(c_1, s_1, A_1, (st, c)) = \left\{ SX \mid \left( \begin{array}{l} \forall\, c_2, s_2, A_2 \bullet \\ \left( \begin{array}{l} (c_1 \mid s_1 \models A_1) \stackrel{st}{\Longrightarrow} (c_2 \mid s_2 \models A_2) \wedge \\ (\exists\, a \bullet c_2 \wedge c) \wedge stable(c_2 \mid s_2 \models A_2) \end{array} \right) \bullet \\ \exists\, iose \in SX \bullet iose \in IOcsinitials^{a}_{st}(c_2, s_2, A_2) \upharpoonright^{a} c \end{array} \right) \right\}$$

*where*

$$stable(c_1 \mid s_1 \models A_1) = \neg\ \exists\, c_2, s_2, A_2 \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2)$$
$$S \upharpoonright^{a} c = \{(se, c \wedge c_1) \mid (se, c_1) \in S \wedge (\exists\, a \bullet c \wedge c_1)\}$$

The *csacceptances* are computed using the *IOcsinitials* after a given stable configuration of the specification. A configuration is stable if no internal silent evolution is possible directly for its action. Only *IOcsinitials* whose constraints are compatible with the constraint of the tested trace are considered. A filter function $\upharpoonright$ is introduced in order to remove unfeasible initials.

The *csacceptances* set defined above is infinite and contains redundant elements since any superset of a *csacceptances* is also a *csacceptances*. A minimal symbolic acceptances set $csacceptances_{min}$ can be defined to avoid this problem. The $csacceptances_{min}$ set after a given *cstrace* must contain exactly one element from each *IOcsinitials* set. Unlike *csacceptances*, the $csacceptances_{min}$ contain only elements that are possible *IOcsinitials*. The $csacceptances_{min}$ is defined as follows:

**Definition 5.4.10.** *for every* $tr \in cstraces(P)$ *we define*

$$csacceptances_{min}(c_1, s_1, A_1, tr) = \bigotimes \left\{ iose \mid \left( \begin{array}{l} \forall\, c_2, s_2, A_2 \mid \\ \left( (c_1 \mid s_1 \models A_1) \stackrel{tr}{\Longrightarrow} (c_2 \mid s_2 \models A_2) \wedge stable(c_2 \mid s_2 \models A_2) \right) \\ \quad \bullet\ iose \in IOcsinitials(c_2, s_2, A_2) \end{array} \right) \right\}$$

The $\bigotimes$ operator defined below is a generalized Cartesian product whose elements are sets, rather than tuples. It takes a set of sets $SX$ as argument, and defines also a set of sets, characterized as follows:

$$\bigotimes SX = \left\{ S_1 \mid \begin{array}{l} (\forall\, S_2 \in SX \bullet S_2 \neq \emptyset \wedge (\exists ! e \in S_2 \bullet e \in S_1)) \\ \wedge\ (\forall\, e \in S_1 \bullet (\exists\, S_2 \in SX \bullet e \in S_2)) \end{array} \right\}$$

The resulting $csacceptances_{min}$ of this definition is minimal (not redundant), but can still be infinite. This can raise from an unbound internal nondeterminism in the system that leads to infinite possibilities. In this case, the set cannot be restricted and all elements must be considered.

The test specification for the deadlocks reduction conformance relation is:

$$\frac{tr \in cstraces(P) \wedge S \in csacceptances_{min}(true, \{\!\{\}\!\}, P, tr)}{Prog(tr, S)} \tag{5.40}$$

The symbolic tests are composed of a first part, which is a (symbolic) trace of $P$, and a second one, which is a choice over the elements of one of the constrained symbolic acceptance set of the trace. The final symbolic tests are obtained from these parts by insertion of the verdict events as in Section 3.2.3.2.

### *csacceptances* generation tactic

test-generation in this case is based on the generation of the $csacceptances_{min}$ set. For a given symbolic trace generated from the specification, the generation of the sets of $csacceptances_{min}$ is performed in three steps. First, all possible stable configurations that can be reached by following the given trace are generated. This is done using the same set of derived rules introduced in section 5.4.2 and used for *csinitials set* generation. In the second step, all possible *IOcsinitials* are generated

for each configuration obtained in the first step. The generation is performed by applying one more time the same set of rules. Finally, the generalized Cartesian product is computed from all resulting *IOcsinitials*. The generation tactic is defined in the following algorithm:

**Data**: k : the maximum length of tests

Generate *cstraces* using trace generation tactic for a length k;

**foreach** *generated trace tr* **do**

  Simplify the test specification using the $csacceptances_{min}$ Definition 5.4.10;

  Generate all stable configurations after *tr* using the derived rules;

  **foreach** *generated stable configuration cf* **do**

    Generate all *IOcsinitials* after this configuration cf using the derived rules;

  **end**

  Introduce the definition of $\bigotimes$ for the resulting set;

  Apply simplification rules to generate the sets $csacceptances_{min}$;

**end**

## 5.5 Test Selection Hypotheses

Symbolic tests cannot be used directly for testing. A finite number of concrete (executable) tests must be instantiated first from these symbolic tests. However, in some situations, no finite number of instances can be found. This problem can result from two cases: (i) either the symbolic test is unbounded, (ii) or the symbolic test accepts an infinite number of instances. Some selection criteria can be used to choose a finite subset of concrete tests. They are formalized as test selection hypotheses: assuming these hypotheses the selected tests form an exhaustive test set [GLG08, CG11].

The length of the symbolic tests can be unbounded if the specification contains loops. A classical selection hypothesis that can be used in this case is the *regularity hypothesis* over values that support a size function. This regularity hypothesis is defined over the traces length. It states that if the SUT behaves correctly for traces shorter than a given length, it will then behave correctly for all the possible traces.

For the case of other value types used to define infinite symbolic tests, other selection criteria are needed to choose a finite subset of concrete tests. A *uniformity hypothesis* can be used to state that the SUT will behave correctly for all the

instances if it behaves correctly for some subset of them. Such a subset can be obtained using on-the-fly constraint solving as, for instance, in [BW07, BW12]

Test selection hypotheses are used explicitly in our test-generation framework *CirTA* (*Circus* Test Automation). A regularity hypothesis is used on traces length, where the maximum regularity length is provided as parameter. This hypothesis is integrated in the test theorem before the generation starts. The trace length limit is then used to stop the traces generation. The generated symbolic traces are then used to generate symbolic tests corresponding to each conformance relation. For each resulting symbolic test, a uniformity hypothesis is stated to extract a *witness* value for each symbolic value in the test. Concrete (*witness*) values are represented by Isabelle *schematic variables* representing arbitrary (but constrained) values.

The regularity and uniformity hypotheses are respectively defined as introduction rules as follows:

$$\frac{\begin{array}{c}[length(tr) < k] \\ \vdots \\ P\ (tr)\end{array} \qquad THYP((\forall\, tr \mid length(tr) < k \bullet P\ (tr)) \rightarrow (\forall\, tr \bullet P\ (tr)))}{\forall\, tr \bullet P\ (tr)}$$

$$\frac{P\ ?x_1 ... ?x_n \qquad THYP((\exists\, x_1, ..., x_n \bullet P\ x_1 ... x_n) \rightarrow (\forall\, x_1, ..., x_n \bullet P\ x_1 ... x_n))}{\forall\, x_1, ..., x_n \bullet P\ x_1 ... x_n}$$

where $P$ is a predicate that characterizes a (symbolic) test case, $tr$ is a (symbolic) trace and $THYP$ is a constant used to preserve test hypotheses from automatic simplifications. Schematic variables are represented in Isabelle by prefixing their names with the ? symbol.

## 5.6 Test Instantiations

The last step of test-generation is the selection of witness values corresponding to schematic variables produced by the uniformity hypothesis. Constraint solvers that are integrated with Isabelle are used for this instantiation, in the same way as what was done in [BW12]. Two kind of solvers can be used: random solvers and SMT solvers. The random constraint solving is performed using QuickCheck, that instantiates randomly the values of the schematic variables. An integration of QuichCheck with the Isabelle simplifier defined for HOL-TestGen can also be used for more efficient random solving. The second kind of integrated constraint solvers are SMT solvers and especially Z3.

An important drawback of the "off-line" test instantiation presented so far, is that it instantiates input but also output values. When executing such a test, the system may return different but correct output values, that might be seen as inconclusive tests. The solution to this problem is to use the so-called "on-line" test instantiation performed at test execution time. In this case, only input values are instantiated and every output value is tested against the constraint to check its validity. If the returned value is correct *w.r.t.* to the constraint, it will be used to simplify the remaining test constraint and the execution continues.

## 5.7 Example

In this section, the test-generation tactics are illustrated using our small example, the *FIG* specification (Fig. 3.1). First, we recall the specification of *FIG*, then the three generation tactics are illustrated. For this example, we use a concrete version of *FIG*, where identifiers are natural numbers:

**channel** $req$ **channel** $ret, out : \mathbb{N}$

**process** $FIG \mathrel{\widehat=}$ **begin**

**state** $S == [\, idS : \mathbb{P}\, \mathbb{N}\,]$

$Init \mathrel{\widehat=} idS := \emptyset$

*Out*

$\Delta S$

$v! : \mathbb{N}$

---

$v! \notin idS$

$idS' = idS \cup \{v!\}$

*Remove*

$\Delta S$

$x? : \mathbb{N}$

---

$idS' = idS \setminus \{x?\}$

• $Init$ ; **var** $v : \mathbb{N}$ •
$(\mu\ X \bullet (req \rightarrow Out\,;\ out!v \rightarrow Skip \ \Box\ ret?x \rightarrow Remove)\,;\ X)$

**end**

### 5.7.1 Generating *cstraces*

For generating the *cstraces* of *FIG*, the test specification (5.37) is used as proof state:

$$\frac{tr \in cstraces(FIG)}{Prog(tr)}$$

The regularity hypothesis is applied for a length less than 4, this results in two subgoals:

$$\frac{length(tr) < 4 \quad tr \in cstraces(FIG)}{Prog(tr)}$$

$$THYP(\frac{\forall\, tr \bullet length(tr) < 4 \wedge tr \in cstraces(FIG) \longrightarrow Prog(tr)}{\forall\, tr \bullet tr \in cstraces(FIG) \longrightarrow Prog(tr)})$$

The first subgoal describes the selected tests using the regularity principle. The second subgoal contains the regularity hypothesis protected using the constant *THYP*. This constant is used to prevent the system from simplifying the hypotheses. The first subgoal is simplified with the definition of *cstraces* (see Definition 5.4.2) and the result is:

$$\frac{length(tr) < 4 \quad \exists\, s_2\, P_2 \bullet (true \mid \{\!|\}\!| \models FIG) \stackrel{tr}{\Longrightarrow} (c_2 \mid s_2 \models P_2)}{Prog(tr)}$$

Applying the elimination rules of the relation "$\Longrightarrow$" leads to three new subgoals:

$$\frac{\exists\, s_2\, P_2 \bullet (true \mid \{\!|\}\!| \models FIG) \stackrel{[]}{\Longrightarrow} (true \mid \{\!|\}\!| \models FIG)}{Prog([])}$$

$$\frac{\begin{array}{c} length(tr) < 4 \quad \exists\, s_2 ... \bullet (true \mid \{\!|\}\!| \models FIG) \stackrel{\varepsilon}{\longrightarrow} (c_1 \mid s_1 \models P_1) \\ \wedge\, (c_1 \mid s_1 \models P_1) \stackrel{tr}{\Longrightarrow} (c_2 \mid s_2 \models P_2) \end{array}}{Prog\,(tr)}$$

$$\frac{\begin{array}{c} length(tr) < 3 \quad \exists\, e\, s_2 ... \bullet (true \mid \{\!|\}\!| \models FIG) \stackrel{e}{\longrightarrow} (c_1 \mid s_1 \models P_1) \wedge \\ (c_1 \mid s_1 \models P_1) \stackrel{tr}{\Longrightarrow} (c_2 \mid s_2 \models P_2) \wedge e \neq \varepsilon \end{array}}{Prog\,(e\#tr)}$$

The first case gives the empty trace, the last two cases introduce a predicate with the transition relation of the operational semantics. The application of the elimination rules of the operational semantics to the second rule attempts to find an applicable transition rule with a label $\varepsilon$. If the system fails to find any applicable rule then the subgoal is removed (false premise). For the third subgoal, the system tries to find a rule that matches the premise and instantiates the unknown variables by their corresponding values. The generation tactic is applied repeatedly until no simplification is possible.

The final proof state represents all the *cstraces* (up to the given length 4) of the *FIG* process:

$Prog\,([])$
$Prog\,([req])$
$Prog\,([ret.a])$
$Prog\,([req, out.a])$
$Prog\,([ret.a, req])$
$Prog\,([ret.a, ret.b])$
$Prog\,([req, out.a, req])$
$Prog\,([req, out.a, ret.b])$
$Prog\,([ret.a, req, out.a])$
$Prog\,([ret.a, ret.b, req])$
$a \neq b \longrightarrow Prog\,([req, out.a, req, out.b])$
$Prog\,([req, out.a, ret.b, req])$
$Prog\,([req, out.a, ret.b, ret.c])$
$Prog\,([ret.a, req, out.a, req])$
$Prog\,([ret.a, ret.b, ret.c])$

The final proof state – corresponding to the test theorem – contains also the subgoal stating the regularity hypothesis.

## 5.7.2 test-generation

### 5.7.2.1 test-generation for traces inclusion

To generate the $\overline{csinitials}$ of $FIG$, the test specification (5.39) is set as proof goal. The system follows the same steps to generate *cstraces* up to length 4. For example, we consider the trace $[req, out.a, req]$ generated earlier and the test theorem is then simplified to:

$$\frac{e \in \overline{csinitials}(P, [req, out.a, req])}{Prog([req, out.a, req]@[e])}$$

The test specification is simplified using the $\overline{csinitials}$ Definition 5.4.6 and the initials sets rules are applied to generate the *csinitials* for this trace. This results in the following subgoal:

$$\frac{e \in \{d.\alpha_0 \mid \neg \bigvee \{d.\alpha_0 \in \{out.b \mid a \neq b\}\})\}}{Prog([req, out.a, req]@[e])}$$

The elements of $\overline{csinitials}$ are then computed using simplification and case-splitting rules. The resulting proof state is the following:

$Prog\,([req, out.a, req, out.a])$
$Prog\,([req, out.a, req, req])$
$Prog\,([req, out.a, req, ret.b])$

Each symbolic test case is composed of the symbolic trace $[req, out.a, req]$ concatenated with a symbolic forbidden continuation. Tests can then be generated from this proof state, this will be shown in section 5.7.3.

#### 5.7.2.2 test-generation for deadlocks reduction

To verify the second conformance relation the *csacceptances* set must be generated for each trace. The test theorem for deadlocks reduction corresponds to (5.40). For example, considering the previous symbolic trace $[req, out.a, req]$ the initial proof goal is simplified to:

$$\frac{S \in csacceptances_{min}(true, \{\!|\}, P, ([req, out.a, req]))}{Prog([req, out.a, req], S)}$$

The *IOcsinitials* sets after this trace are then generated using the corresponding rules. The result is given in the following subgoal:

$$\frac{S \in \bigotimes \{\{out?b \mid a \neq b\}\}}{Prog([req, out.a, req], S)}$$

The minimal *csacceptances* sets are then computed for this trace by simplifying the general Cartesian product. The final proof state for this case contains the following subgoal:

$$Prog\,([req, out!a, req], \{out?b \mid a \neq b\})$$

### 5.7.3 Test instantiation and presentation

The obtained symbolic tests must be instantiated in order to be executed. Explicit test verdicts are also introduced in the final representation of tests in form of *Circus* prefixed actions. The verdict of the execution of a test is given by the last produced verdict event after this execution.

Considering the same symbolic trace treated earlier $[req, out.a, req]$, the uniformity hypothesis rule is applied for each resulting test. Constraint solving is then applied to instantiate schematic variables in the test definitions. The resulting proof state contains 4 concrete tests and 4 uniformity hypotheses expressed as follows:

**Traces inclusion**

$Prog\,([req, out.1, req, out.1])$
$THYP(\exists\, a \bullet Prog\,([req, out.a, req, out.a]) \rightarrow \forall\, a \bullet Prog\,([req, out.a, req, out.a]))$

$Prog\,([req, out.1, req, req])$
$THYP(\exists\, a \bullet Prog\,([req, out.a, req, req]) \rightarrow \forall\, a \bullet Prog\,([req, out.a, req, req]))$

$Prog\,([req, out.1, req, ret.2])$
$THYP(\exists\, a\ b \bullet Prog\,([req, out.a, req, ret.b]) \rightarrow \forall\, a\ b \bullet Prog\,([req, out.a, req, ret.b]))$

**Deadlocks reduction**

$$Prog\,([req, out!1, req], \{out?b \mid 1 \neq b\})$$
$$THYP(\exists\, a\ b \bullet Prog\,([req, out.a, req], \{out?b \mid a \neq b\}) \rightarrow \forall\, a\ b \bullet Prog\,([req, out.a, req], \{out?b \mid a \neq b\}))$$

The different test verdicts are inserted in the corresponding places as seen in section 5.4.2 and section 5.4.3 and the tests are presented in forms of prefixed actions. The 4 generated tests are given as follows:

**Traces inclusion**

$$inc \rightarrow req \rightarrow inc \rightarrow out.1 \rightarrow inc \rightarrow req \rightarrow pass \rightarrow out.1 \rightarrow fail \rightarrow Stop$$

$$inc \rightarrow req \rightarrow inc \rightarrow out.1 \rightarrow inc \rightarrow req \rightarrow pass \rightarrow req \rightarrow fail \rightarrow Stop$$

$$inc \rightarrow req \rightarrow inc \rightarrow out.1 \rightarrow inc \rightarrow req \rightarrow pass \rightarrow ret.2 \rightarrow fail \rightarrow Stop$$

**Deadlocks reduction**

$$inc \rightarrow req \rightarrow inc \rightarrow out.1 \rightarrow inc \rightarrow req \rightarrow fail \rightarrow out?x : x \neq 1 \rightarrow pass \rightarrow Stop$$

## 5.8 Conclusions

We have described in this section the *CirTA* test-generation framework for *Circus* specifications. Test definitions were introduced in [CG11], which provides a basis for the development of various testing strategies from *Circus* specifications. Our approach to test-generation is based on the logical environment of Isabelle/HOL. In Chapter 4 we presented formalizations of UTP and *Circus* as Isabelle theories. These theories provide a framework for *CirTA*.

The test definitions are based on the operational semantics of *Circus* defined as a labeled transition relation. We provide a formalization of the rules of the operational semantics on top of Isabelle/*Circus* and explain some representation choices. Using these rules, we derive a number of other rules that are important for test-generation. Test definitions are introduced and then automatic generation tactics are defined as a proof method in Isabelle.

Isabelle/HOL is a mature theorem prover and easily supports our requirements for add-on tools for symbolic computation, but substantial efforts had to be invested for building our formal testing environment nonetheless. With regard to the experience of the last 10–20 years of the interactive theorem proving community, this initially steep ascent is in fact quite common, and we can anticipate eventual pay-off for more complex examples at the next stage. HOL as a logic opens a wide

space of rich mathematical modeling, and Isabelle/HOL as a formal tool environment supports many mathematical domains by proof tools, say for simplification and constraint solving. Many of these Isabelle tools already incorporate other external proof tools, such as Z3. Thus we take advantage from this rich collection of formal reasoning tools for our particular application of *Circus* conformance testing, and exploit the full potential of theorem prover technology for our work.

For instance, the choice of a shallow symbolic representation using Isabelle's symbolic facilities for test definitions and generation has turned out to be judicious and powerful since it avoids lots of heavy technicalities (in particular in the management of symbolic alphabets).

A small example is given at the end of this chapter to illustrate the test definition and generation process. Test specifications are stated as proof goals, and test selection hypotheses are expressed as introduction rules. In this first presentation we consider off-line testing but it is clear that more realistic applications require on-line testing.

In Chapter 6, the test-generation environment is applied on a larger case study. A *Circus* specification is written for a message monitoring system, implemented as a multi-queue. The test-generation tactics are used to generate tests from this specification. A Java implementation of the monitoring system is tested using the generated tests.

CHAPTER 6

# Case Study

## Contents



## 6.1 Introduction

In this section we present a case study on *Circus*-based testing using *CirTA*. The studied system is a message monitoring module implemented as a FIFO queue. This module is a part of a remote medical monitoring server, which is described in section 6.2. In section 6.3, an abstract *Circus* specification of the queue module is provided. This section contains also the Isabelle/*Circus* specification used by the test generation system. This specification is used as a basis for test generation and execution as described in section 6.4. The generated tests are used to test a concrete

Java implementation of the queue. Finally, some conclusions and discussions on the case study are given in section 6.5.

## 6.2 Remote Monitoring System

Our case study is a part of the remote monitoring system used in a worldwide healthcare network developed by BIOTRONIK SE & Co. KG[1]. The network connects a variety of remote devices that are in general pacemaker controllers or similar medical devices. The automatic monitoring system keeps track of the status of all connected devices that regularly send diagnostic, therapeutic, and technical data on the current clinical status of the patients.

The monitoring system collects a huge number of messages received from the different patients' devices. The messages are then routed to their corresponding processing services in order to be processed. The routing policy is a bit particular and depends on the nature of the message and the type of the device. This routing policy is very critical and must be correctly implemented in the monitoring system. A wrong message routing may lead to information misinterpretation that can be risky for the patients health.

An overview of the remote monitoring system (also called *home monitoring*) is given in Figure 6.1. The whole system connects different local patients devices and physicians control systems. Patients devices periodically send messages to the remote monitoring system describing its status. The monitoring system synthesizes the messages and send health reports to the physicians control systems.

Figure 6.1: Remote monitoring system overview

The remote monitoring system is composed essentially of a queue module and a set of processing services. The different message manipulations and routing operations are carried out by the queue module. Each message is characterized with a device identifier and its actual content. The queue receives, stores, and then assigns messages to the corresponding processing services. The main operations that can be performed by the queue are:

PUT: The queue receives the messages using the PUT operation. These messages are stored in the queue with the *new* status.

[1] www.biotronik.com

GET: The processing services can retrieve *new* messages from the queue in order to process them. During the processing time, the message is marked as *active*.

FINISH: When a processing service successfully completes a message processing, it informs the queue using the FINISH operation. This *active* message is then completely removed from the queue.

RENEW: An *active* message can be reconsidered as a *new* message if its processing was not successful (after some time). This can be done using the RENEW operation and the message can be assigned again to another processing service.

DISCARD: An *active* message can be removed even if its computation was not finished successfully using the DISCARD operation. This is needed if several processing services attempt to process this message but never succeed. The reason is generally that the message is erroneous.

DELETE: In some situations, some messages are not required anymore. These messages can then be removed directly from the queue using the DELETE operation.

Figure 6.2 gives a representation, as a state diagram, of the life cycle of a message in a queue. First, the *new* message is added using the PUT operation. Once added, the *new* message can either be retrieved by a processing service or completely deleted. In the first case, and after the GET operation, the message becomes *active*. An *active* message can be processed successfully and then be removed from the queue using the FINISH operation. It can also be reconsidered for processing using the RENEW operation. Erroneous messages can be discarded and removed from the queue using the DISCARD operation. Obsolete or superfluous messages can also be removed using the DELETE operation.

All these operations operate on *topics*, that define the kind of job described in the message. For example, one topic may be sending faxes, while another one can be medical patient data that was gathered by a pacemaker. Each processing service is defined for a given kind of messages, *i.e.* topic, and thus receives only messages of this topic. The ordering of messages of different topics is irrelevant. Only messages of the same topic are ordered since they can be processed by the same processing services. In a more abstract view, the queue can be seen as a multi-queue, where each individual queue deals with a different topic.

As said above, device identifiers are usually associated with the messages. The ordering of the messages depends on their source. Two messages sent from the same device should not be processed simultaneously; the processing order follows the sending order. This order is important since the processing of the later message may depend on the results of the processing of the first one. Messages that come

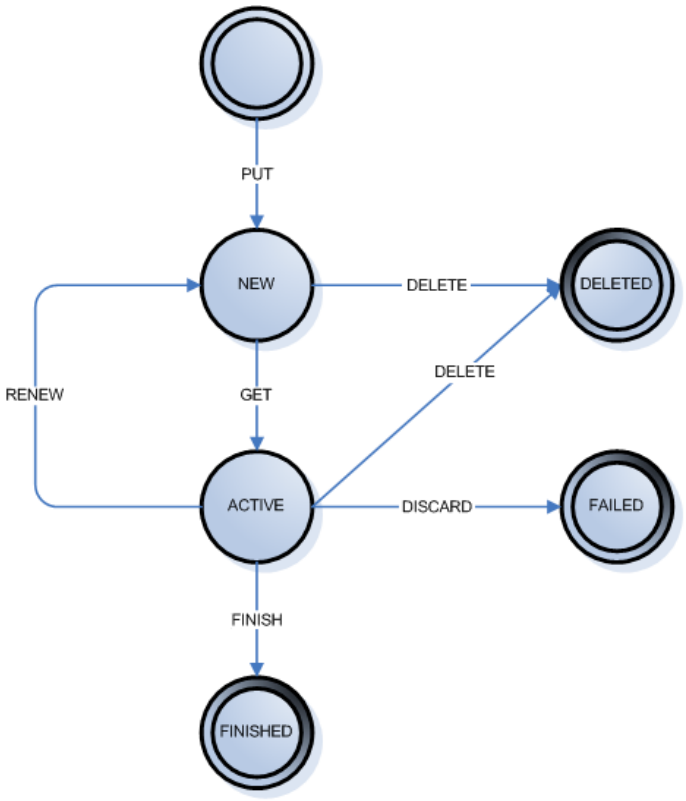



Figure 6.2: Messages life cycle

from different sources are processed following the arrival order following a FIFO discipline.

## 6.3 Abstract Queue Specification

In this section, we provide an abstract *Circus* specification of the queue module. For the sake of simplicity, some general considerations and abstractions are stated. The first one is that only one topic is considered; all messages are assumed to belong to the same topic. This simplification will reduce the number of generated tests by avoiding redundancy. The second consideration is about message identifiers and contents: they are considered as natural numbers in this specification. A last restriction is made on the operations that are supported in the specification. Only three operations are considered: PUT, GET and FINISH. The other operations depend more on the processing services behavior than on the queue itself; for instance, on extra information (time, message content ...) that is not considered in this abstract specification. This test objective has been delimited by the available knowledge of

the environment of the system. It is clear that it limits the overall coverage of the generated tests to the considered operations.

We start the specification by the definition of the process channels. Each queue operation corresponds to a channel in our specification. Three channels are defined *get*, *put* and *finish*, all of type $\mathbb{N} \times \mathbb{N}$ (pairs of natural numbers).

$$\textbf{channel}\ get, put, finish : \mathbb{N} \times \mathbb{N}$$

The type of the channels describes the structure of the communicated messages. The first element of the message is the device identifier and the second element represents the actual content of the message.

Once the channels are defined, we introduce the internal state of the queue.

$$\textbf{state}\ QueueState\ ==\ [\, new : \mathrm{seq}(\mathbb{N} \times \mathbb{N});\ active : \mathbb{P}(\mathbb{N} \times \mathbb{N})\,]$$

The state contains two fields: *new* and *active*. The first field *new* is the sequence of messages that have been newly added to the queue. These messages are ordered following the order of their arrival. The *active* field gives the set of messages that have been retrieved by a processing service in order to be processed.

The global behavior of the process can be described as follows:

$$\textbf{var}\ x, y : \mathbb{N} \times \mathbb{N} \bullet InitQueue\ ;\ (\,\mu\, X \bullet (\, Put \,\Box\, Get\ \Box\, Finish\,)\ ;\ X\,)$$

First, the queue internal state is initialized using the *InitQueue* action. Then, the queue offers recursively a choice over the three actions *Put*, *Get* and *Finish*. These three actions corresponds to the three queue operations of the same names.

The initialization is given by the *InitQueue* schema expression. The *new* field is initialized by an empty sequence and the *active* field by an empty set.

*InitQueue*

$$QueueState'$$

---

$$new' = \langle\,\rangle \land active' = \emptyset$$

The *Put* action corresponds to the PUT operation of the queue. Whenever the queue receives a message on the *put* channel, it stores it in the *new* messages sequence.

$$Put\ \mathrel{\widehat{=}}\ put\,?x\ \rightarrow\ AddNew$$

The *AddNew* schema appends the message at the end of the *new* sequence.

*AddNew*

$\Delta QueueState$
$x? : \mathbb{N} \times \mathbb{N}$

---

$new' = new \frown \langle x? \rangle$

The second action *Get* corresponds to the GET operation of the queue. If messages are available, processing services can retrieve them via the *get* channel. The availability of messages is ensured by a guard stating that it exists at least one *new* message whose identifier is not in the *active* set. The returned message is chosen using the *Choose* action and the message activation is done by the *Activate* action.

$$Get \;\widehat{=}\; (\operatorname{dom} new \setminus \operatorname{dom} active \neq \emptyset) \,\&\, (Choose;\; get!y \;\rightarrow\; Activate)$$

The *Choose* action is defined by a schema that returns a message from the sequence of *new* messages. The chosen message is the first element of this sequence whose identifier is not presently processed. This will prevent two messages that have the same identifier from being processed in the same time.

*Choose*

$QueueState$
$y! : \mathbb{N} \times \mathbb{N}$

---

$\operatorname{dom} new \setminus \operatorname{dom} active \neq \emptyset$
$y! = head\,(new \rhd \{a, b : \mathbb{N} \mid a \notin \operatorname{dom} active \bullet (a, b)\})$

After sending the message to a processing service, the queue must mark this message as *active*. This means that the message is added to the *active* set and its first occurrence is removed from the *new* sequence. Only the first occurrence is removed to avoid deleting intentionally duplicated messages.

*Activate*

$\Delta QueueState$
$y? : \mathbb{N} \times \mathbb{N}$

---

$new' = RemoveFirst(new, y?) \land active' = active \cup \{y?\}$

In order to remove the first occurrence of a message from a sequence of messages, we introduce a recursive function *RemoveFirst*.

$RemoveFirst : \operatorname{seq}(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \rightarrow \operatorname{seq}(\mathbb{N} \times \mathbb{N})$

---

$RemoveFirst(\langle\,\rangle, n) = \langle\,\rangle$
$\forall\, S : \operatorname{seq}_1(\mathbb{N} \times \mathbb{N}), n : (\mathbb{N} \times \mathbb{N}) \bullet$
$RemoveFirst(S, n) =$
$\quad \mathbf{if}_Z\ head\ S = n\ \mathbf{then}\ tail\ S\ \mathbf{else}\ \langle head\ S \rangle \frown RemoveFirst(tail\ S, n)$

The last action is *Finish*, which corresponds to the FINISH operation of the queue. Whenever a processing service finishes the processing of an active message, this one is completely removed from the queue. Note that only *active* messages can be returned via the *finish* channel.

$$Finish \mathrel{\widehat=} finish?x \in active \rightarrow Remove$$

The *Remove* schema removes the returned message from the *active* messages set.

*Remove*

$$\Delta QueueState$$
$$x? : \mathbb{N} \times \mathbb{N}$$
$$active' = active \setminus \{x?\}$$

The full *Circus* specification of the abstract queue process is given in the following.

**channel** $get, put, finish : \mathbb{N} \times \mathbb{N}$

**process** $Abstract\_Queue \mathrel{\widehat=}$ **begin**

**state** $QueueState == [\, new : \mathrm{seq}(\mathbb{N} \times \mathbb{N});\ active : \mathbb{P}(\mathbb{N} \times \mathbb{N}) \,]$

$$RemoveFirst : \mathrm{seq}(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N}) \rightarrow \mathrm{seq}(\mathbb{N} \times \mathbb{N})$$
$$RemoveFirst(\langle\,\rangle, n) = \langle\,\rangle$$
$$\forall\, S : \mathrm{seq}_1(\mathbb{N} \times \mathbb{N}), e : (\mathbb{N} \times \mathbb{N})\ , n : (\mathbb{N} \times \mathbb{N}) \bullet$$
$$RemoveFirst(\langle e \rangle \frown S, n) =$$
$$\quad \mathbf{if}_Z\ e = n\ \mathbf{then}\ S\ \mathbf{else}\ \langle e \rangle \frown RemoveFirst(S, n)$$

*InitQueue*

$$QueueState'$$
$$new' = \langle\,\rangle \land active' = \emptyset$$

*AddNew*

$$\Delta QueueState$$
$$x? : \mathbb{N} \times \mathbb{N}$$
$$new' = new \frown \langle x? \rangle$$

*Choose*

$$QueueState$$
$$y! : \mathbb{N} \times \mathbb{N}$$
$$\text{dom } new \setminus \text{dom } active \neq \emptyset$$
$$y! = head\,(new \upharpoonright \{a, b : \mathbb{N} \mid a \notin \text{dom } active \bullet (a, b)\})$$

*Activate*

$$\Delta QueueState$$
$$y? : \mathbb{N} \times \mathbb{N}$$
$$new' = RemoveFirst(new, y?) \land active' = active \cup \{y?\}$$

*Remove*

$$\Delta QueueState$$
$$x? : \mathbb{N} \times \mathbb{N}$$
$$active' = active \setminus \{x?\}$$

$$Put \mathrel{\widehat=} put?x \rightarrow AddNew$$
$$Get \mathrel{\widehat=} (\text{dom } new \setminus \text{dom } active \neq \emptyset)\,\&\,(Choose;\ get!y \rightarrow Activate)$$
$$Finish \mathrel{\widehat=} finish?x \in active \rightarrow Remove$$
$$\bullet\ \textbf{var}\ x, y : \mathbb{N} \times \mathbb{N} \bullet InitQueue\,;\ (\,\mu\, X \bullet ((\,Put \,\Box\, Get\,) \,\Box\, Finish\,)\,;\ X\,)$$
**end**

Using the syntax of Isabelle/*Circus* given in Appendix A.3, we introduce in the *CirTA* system a formalization of the abstract queue process. First, we define the `RemoveFirst` function, that removes the first occurrence of an element from a list.

```
1 fun RemoveFirst::(nat×nat) list ⇒(nat×nat) ⇒(nat×nat) list where
2  RemoveFirst [] _ = []
3 | RemoveFirst (a#l) n = (if a=n then l else (a#(RemoveFirst l n)))
```

The definition of the `Abstract_Queue` process is then introduced using the `circusprocess` command. This definition introduces system variables (alphabet), the state definition, channels declaration, schema expressions and actions definitions. The system variables (alphabet) and the state variables are introduced first. The alphabet contains two variables `x` and `y` which are pairs of natural numbers. The state contains two fields, `new`: a list of pairs and `active`: a set of pairs.

```
1   alphabet = [x::nat×nat, y::nat×nat]
2   state = [new::(nat×nat) list, active::(nat×nat) set]
```

The channels are declared by giving their names and types. Three channels are defined `get`, `put` and `finish` communicating the same type: pairs of naturals.

```
channel = [get nat×nat, put nat×nat, finish nat×nat]
```

All schema expressions are introduced using the `schema` command. The first expression is `InitQueue` which initializes the queue state as follows:

```
schema InitQueue = new' = [] ∧active' = {}
```

The `AddNew` schema appends the new element contained in the variable `x` at the end of the `new` list. The `@` operator is the list concatenation operator and `[x]` is the singleton list that contains only the variable `x`.

```
schema AddNew = new := new@[x]
```

The `Choose` schema returns the first *activable* element of the `new` list. An *activable* element is a message whose identifier is not in the `active` set. The list of *activable* elements is retrieved by filtering `new` list using the HOL `filter` function. This function accepts two arguments: a filter function and a source list. It returns a list containing the elements of the source list that satisfy the filter function.

```
schema Choose =     (∃a∈set new. fst a ∉fst ‘ active) ∧
          y := hd (filter (λb.∀a∈active. fst a ≠fst b) new)
```

An element is activated using the `Activate` schema operations. The first occurrence of this element, contained in the variable `y`, is removed from the `new` list using `RemoveFirst`. This element is added to the `active` element set.

```
schema Activate = new' = RemoveFirst new y ∧active' = active ∪{y}
```

An element can be completely removed from the queue using the `Remove` schema.

```
schema Remove = active := active - {x}
```

The `Abstract_Queue` process contains three actions: `Put`, `Get` and `Finish`. The `Put` action receives a message on the `put` channel then calls the `AddNew` schema.

```
action Put = put?x →(Schema AddNew)
```

The second action `Get` is a guarded action. It is applicable only if there exist elements in `new` whose identifiers are not in `active`. In this case, the `Choose` schema is called to choose the oldest of these element. This element is the returned via the `get` channel and activated using the `Activate` schema.

```
action Get =
    (λ A. (fst‘(set (new A))) - (fst‘(active A)) ≠{}) &
              ((Schema Choose); get!y →(Schema Activate))
```

The last action is `Finish`. It completely removes from the queue the active element retrieved on the `finish` channel.

```
action Finish = finish?x∈active →(Schema Remove)
```

The main action of the process is composed of an initialization, then a recursive choice over the three defined actions.

```
(Schema InitQueue); μX •(((Put □ Get) □ Finish); X)
```

The full definition of the `Abstract_Queue` process in Isabelle/*Circus* is given by the following command:

```
circusprocess Abstract_Queue =
  alphabet = [x::nat×nat, y::nat×nat]
  state = [new::(nat×nat) list, active::(nat×nat) set]
  channel = [get nat×nat, put nat×nat, finish nat×nat]
  schema InitQueue = new' = [] ∧active' = {}
  schema AddNew = new := new@[x]
  schema Choose =     (∃a∈set new. fst a ∉fst ` active) ∧
           y := hd (filter (λb.∀a∈active. fst a ≠fst b) new)
  schema Activate = new' = RemoveFirst new y ∧active' = active ∪{y}
  schema Remove = active := active - {x}
  action Put = put?x →(Schema AddNew)
  action Get =
       (λ A. (fst`(set (new A))) - (fst`(active A)) ≠{}) &
                 ((Schema Choose); get!y →(Schema Activate))
  action Finish = finish?x∈active →(Schema Remove)
  where (Schema InitQueue); μX •(((Put □Get) □ Finish); X)
```

## 6.4 Testing the Queue Implementation

The queue is implemented in Java and integrated to the whole remote monitoring system. In order to test this implementation, JUnit testing facilities are used for test execution. Starting from the queue specification, tests are generated then translated into JUnit test cases. The resulting test cases are then directly executed on the given implementation.

### 6.4.1 Test generation

The test generation for the Queue process is done in two steps. First, all possible traces (up to a given length) are generated. Then, for each generated trace, two test sets are generated, one for trace inclusion and one for deadlock reduction. The symbolic generated tests are then transformed into instantiated tests via some HOL-TestGen's method called `gen_test_data`, and then into executable tests that will be exercised against the system (see subsection 6.4.2).

#### 6.4.1.1 Trace generation

As seen in section 5.4, a test specification is stated as a proof goal describing the traces to generate. This test specification, in our case, is given by the following formula:

```
1  tr ∈cstraces Abstract_Queue ⟹ prog tr
```

The trace generation tactic is invoked on this proof goal, using different trace lengths. A first (expected) drawback of the present trace generation tactic is the lack of efficiency due to the fact that we generate exhaustively. In our case, the generation time grows exponentially *w.r.t.* the trace length. For a length of 4, the generation takes more than 20 seconds against a maximum of 5 seconds for shorter traces. For traces of length 5, the generation time is around 5 minutes. The generation is done using Isabelle2011-1 running on a computer working on Windows 7. The computer has an 8-core processor (Intel i7 2600) and 6 GB of RAM.

This is due also to the heavy machinery used for trace generation and also to the multiple silent transitions of the operational semantics. A characteristic of our specification is that, after the initialization, it behaves in a recursive way. We can take advantage of this characteristic to improve the trace generation efficiency by factorizing the generations steps. During one recursion, different silent transitions are performed, in addition to one communication transition. All these transitions can be factorized in a one-step transition that covers the silent and the communication transitions. A specific rule for this transition called `OneStep` was proved from the operational semantics.

Using the `OneStep` rule, the exhaustive generation time is reduced. For a length of 5, the generation takes less than 10 seconds and for 6 less than 25 seconds. For longer traces, the generation time grows also exponentially, *e.g.* 5 minutes for traces of length 7. For traces of length 8 or less, the generation time is around 1 hour, producing more than 2560 different traces.

Thus we were led to limit the length of generated traces to 6 and thus the generated tests will have a maximum length of 7. This limit is chosen only for practical reasons, due to the current bad performance, and the number of traces: the number of generated traces using this limit is around 150. Thus the whole test generation will take an important execution time. In the near future, we plan to consider more restrictive selection hypothesis in order to focus on interesting and longer selected tests. Moreover, the remaining manual tasks must be automated to reduce the test construction time.

Examples of generated traces are the following:

```
1 ⋀a b aa ba ab bb ac bc.
2         prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc)]
3 ⋀a b aa ba ab bb.
```

```
4         prog [put (a, b), put (aa, ba), put (ab, bb), get (a, b)]
5 ⋀a b aa ba ab bb.
6         prog [put (a, b), put (aa, ba), put (ab, bb)]
7 ⋀a b aa ba.
8         prog [put (a, b), put (aa, ba), get (a, b), finish (a, b)]
9 ⋀a b aa ba ab bb.
10        prog [put (a, b), put (aa, ba), get (a, b), put (ab, bb)]
11 ...
```

As said above, for practical reasons the length limit considered for the moment is 6. A regularity hypothesis is stated on the length of traces. This regularity hypothesis is given as follows:

```
1 THYP ((length tr ≤ 6  ⟶ prog tr) ⟶ (∀ tr. prog tr))
```

#### 6.4.1.2 Test generation for trace inclusion

For each trace generated in the first step, different test cases are generated to test the trace inclusion relation. A test specification is stated as a proof goal in order to start the generation. The complete test specification is given as follows:

```
1   tr ∈ cstraces Abstract_Queue ∧ e ∈ csinitialsb Abstract_Queue tr
2         ⟹   prog tr@[e]
```

The trace generation tactic instantiates the variable `tr` of this test specification to all the possible traces. This results into different test specifications each associated to a different trace. The initials generation tactic is used to generate the corresponding initials for each test specification. The brute force combination of these two tactics would be very slow and require a lot of resources. More efficient combinations will be discussed in the conclusion of this chapter. For the moment, we avoid this by separating (manually) the different test specifications into different theory files.

A test specification is written for each trace; it describes the set of non-initials after this trace. The test cases are then retrieved by unfolding the definition of `csinitialsb` in the test specification, then simplifying the resulting proof goal. Like for traces, the generation of initials is also slow when using the operational semantics rules directly. A factorized version of the initials generation rules was also derived and proved.

As an example, let us consider the generated trace `[put (a, b), put (aa, ba), put (ab, bb), put (ac, bc), put (ad, bd), put (ae, be)]`. This trace is used to illustrate the test generation tactic and its results. The test specification corresponding to this trace is given in the following:

```
1 ⋀a b aa ba ab bb ac bc ad bd ae be.
```

```
e∈csinitialsb Abstract_Queue [put (a, b), put (aa, ba), put (ab, bb)
    , put (ac, bc), put (ad, bd), put (ae, be)] ⟹
        prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
            put (ad, bd), put (ae, be), e]
```

The result of the test generation tactic is the list of the possible tests, defined by the trace and a non initial element. In this case, three different tests are generated, each one associated to a constraint (`af` $\neq$ `a` and `bf` $\neq$ `b`).

```
⋀a b aa ba ab bb ac bc ad bd ae be af bf. af ≠ a ⟹
        prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
            put (ad, bd), put (ae, be), get (af, bf)]
```

```
⋀a b aa ba ab bb ac bc ad bd ae be af bf. bf ≠ b ⟹
        prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
            put (ad, bd), put (ae, be), get (af, bf)]
```

```
⋀a b aa ba ab bb ac bc ad bd ae be af bf.
        prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
            put (ad, bd), put (ae, be), finish (af, bf)]
```

These tests are represented in a symbolic way, using symbolic HOL variables (*e.g.* `a, b, ba ...`). To obtain concrete finite test cases, some selection hypotheses must be stated on the symbolic tests. We reused in this step the `gen_test_cases` method of the HOL-TestGen system [BW12]. This method makes more simplifications on the current symbolic tests, especially on the constraint part. It applies also a uniformity hypothesis on the simplified symbolic tests and returns concrete test cases. Concrete values are represented by schematic variables (*e.g.* `?X32X18`) which are also constrained. These schematic variables can be instantiated by any values satisfying the constraints. The resulting *schematic* test cases are presented as follows:

```
?X32X18 ≠  ?X44X30 ⟹
prog [put (?X44X30, ?X43X29), put (?X42X28, ?X41X27), put (?X40X26,
  ?X39X25), put (?X38X24, ?X37X23), put (?X36X22, ?X35X21),
    put (?X34X20, ?X33X19), get (?X32X18, ?X31X17)]
```

```
?X16X17 ≠?X28X29 ⟹
prog [put (?X29X30, ?X28X29), put (?X27X28, ?X26X27), put (?X25X26,
  ?X24X25), put (?X23X24, ?X22X23), put (?X21X22, ?X20X21),
    put (?X19X20, ?X18X19), get (?X17X18, ?X16X17)]
```

```
prog [put (?X14X29, ?X13X28), put (?X12X27, ?X11X26), put (?X10X25,
  ?X9X24), put (?X8X23, ?X7X22), put (?X6X21, ?X5X20),
    put (?X4X19, ?X3X18), finish (?X2X17, ?X1X16)]
```

In addition to the *schematic* test cases, a uniformity hypothesis, corresponding to the constraint, is stated for each test case. The uniformity is applied on all the symbolic variables of the symbolic test, so only one hypothesis is associated to each symbolic test. The resulting proof state contains then the following hypotheses:

```
1 THYP ((∃x xa xb xc xd xe xf xg xh xi xj xk xl xm.
2         xa ≠xm ∧prog [put (xm, xl), put (xk, xj), put (xi, xh),
3          put (xg, xf), put (xe, xd), put (xc, xb), get (xa, x)]) ⟶
4       (∀x xa xb xc xd xe xf xg xh xi xj xk xl xm.
5         xa ≠xm ⟶prog [put (xm, xl), put (xk, xj), put (xi, xh),
6          put (xg, xf), put (xe, xd), put (xc, xb), get (xa, x)]))
```

```
1 THYP ((∃x xa xb xc xd xe xf xg xh xi xj xk xl xm.
2          x ≠xl ∧prog [put (xm, xl), put (xk, xj), put (xi, xh),
3           put (xg, xf), put (xe, xd), put (xc, xb), get (xa, x)]) ⟶
4       (∀x xa xb xc xd xe xf xg xh xi xj xk xl xm.
5          x ≠xl ⟶prog [put (xm, xl), put (xk, xj), put (xi, xh),
6           put (xg, xf), put (xe, xd), put (xc, xb), get (xa, x)]))
```

```
1 THYP ((∃x xa xb xc xd xe xf xg xh xi xj xk xl xm.
2        prog [put (xm, xl), put (xk, xj), put (xi, xh), put (xg, xf),
3          put (xe, xd), put (xc, xb), finish (xa, x)]) ⟶
4      (∀x xa xb xc xd xe xf xg xh xi xj xk xl xm.
5        prog [put (xm, xl), put (xk, xj), put (xi, xh), put (xg, xf),
6          put (xe, xd), put (xc, xb), finish (xa, x)]))
```

In order to be executed, the *schematic* test cases must be instantiated with concrete values. For this, we use also a HOL-TestGen's method called `gen_test_data`. This method uses smt solvers (*e.g.* Z3) to instantiate concrete values for the schematic variables. The resulting test cases of our example are given as follows:

```
1 prog [put (3, 1), put (0, 0), put (0, 2), put (0, 1), put (5, 3),
2               put (1, 4), get (0, 1)]
```

```
1 prog [put (0, 0), put (3, 1), put (0, 1), put (0, 0), put (0, 0),
2               put (0, 0), get (3, 2)]
```

```
1 prog [put (1, 2), put (1, 6), put (1, 0), put (2, 0), put (2, 4),
2               put (0, 4), finish (0, 1)]
```

#### 6.4.1.3 Test generation for deadlock reduction

Similarly to what is done for the trace inclusion relation, each generated trace is used to generate the corresponding tests *w.r.t.* the deadlock reduction conformance

relation. The test specification corresponding to all the possible traces is given as follows:

```
1  tr ∈ cstraces Abstract_Queue ∧ e ∈ csacceptances Abstract_Queue tr
2        ⟹ prog tr e
```

The trace generation tactic presented earlier instantiates the variable `tr` to the possible symbolic traces. As seen for trace inclusion, this results in different test specifications corresponding to each trace. The acceptances generation tactic is then applied automatically to all the resulting cases. The combination of the trace generation tactic and the test generation tactic (including acceptances) would make the whole generation fully automatic. Unfortunately, as for trace inclusion, such a generation would be very slow for exhaustive tests due to the huge size of the resulting proof state.

For the moment, in order to avoid these problems and to introduce some parallelization, each test specification (corresponding to one possible trace) is treated separately. We split (manually) the proof goal resulting from the trace generation tactic to different test specifications, written in different theory files. If we consider for example the trace `[put (a, b), put (aa, ba), put (ab, bb), put (ac, bc), put (ad, bd), put (ae, be)]`, then the corresponding test specification is given by the following:

```
1 ⋀a b aa ba ab bb ac bc ad bd ae be.
2 e ∈ csacceptances Abstract_Queue [put (a, b), put (aa, ba),
3        put (ab, bb), put (ac, bc), put (ad, bd), put (ae, be)] ⟹
4 prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
5        put (ad, bd), put (ae, be)] e
```

The acceptances generation tactic is used to retrieve the corresponding acceptance sets. The resulting proof goal contains in each subgoal the variable `prog` associated to a trace and an acceptance set. As for the trace refinement relation, some factorized rules were derived and used in the generation in order to reduce the generation time. The previous example produces one test case presented as follows:

```
1 ⋀a b aa ba ab bb ac bc ad bd ae be af bf.
2 prog [put (a, b), put (aa, ba), put (ab, bb), put (ac, bc),
3   put (ad, bd), put (ae, be)] {get (a, b), put (af, bf)}
```

The generated symbolic acceptance set is finite in this case, due to the structure of the specification. This is the case for all the generated tests. As a consequence, test selection, instantiation and translation are not very complicated.

Following the same strategy as for trace inclusion, HOL-TesGen is reused for test selection and instantiation. First, the `gen_test_cases` method is applied on the proof goal. This method applies some simplifications and then states a uniformity

hypothesis on the symbolic variables of the test. This results on the following *schematic* test case:

```
prog [put (?X14X45, ?X13X44), put (?X12X43, ?X11X42), put (?X10X41,
  ?X9X40), put (?X8X39, ?X7X38), put (?X6X37, ?X5X36), put (?X4X35,
   ?X3X34)] {get (?X14X45, ?X13X44), put (?X2X33, ?X1X32)}
```

Besides the test case, the following uniformity hypothesis is added to the proof goal:

```
THYP ((∃x xa xb xc xd xe xf xg xh xi xj xk xl xm.
   prog [put (xm, xl), put (xk, xj), put (xi, xh), put (xg, xf),
     put (xe, xd), put (xc, xb)] {get (xm, xl), put (xa, x)}) ⟶
       (∀x xa xb xc xd xe xf xg xh xi xj xk xl xm.
   prog [put (xm, xl), put (xk, xj), put (xi, xh), put (xg, xf),
     put (xe, xd), put (xc, xb)] {get (xm, xl), put (xa, x)}))
```

Finally, and in order to make the tests executable, the *schematic* test cases are instantiated to concrete tests. As in the previous subsection, this is done using the `gen_test_data` method provided by HOL-TestGen. For our example, the resulting concrete test obtained by this method is given by:

```
prog [put (1, 1), put (10, 5), put (0, 0), put (1, 3), put (0, 0),
          put (1, 1)] {get (1, 1), put (0, 2)}
```

### 6.4.2 Test execution

In order to execute the concrete tests against the provided Java implementation of the queue, these test cases must be expressed in terms of JUnit test methods. Each event of the trace is translated to a call to the corresponding method in the implementation. The execution is then done directly in the Eclipse platform using JUnit testing facilities.

#### 6.4.2.1 Trace inclusion

For the first conformance relation, the concrete tests generated previously are automatically translated into Java methods. This translation is done using a new method called `export_test_file` that we have developed for this purpose. The translation (ML) method implements some translation rules for each event of the concrete tests. Two cases can be distinguished: the trace events and the last event that represents a non initial event.

For the trace events, the translation is straightforward, `put` and `finish` events are translated directly to the corresponding methods. The `get` event is translated into a call to the corresponding method, followed by a check of the resulting value.

The call of these methods may fail. This is detected by an exception or by a wrong result returned by `get`. If the call to these methods fails, the test is considered inconclusive. A message is printed in this case and the test execution is aborted.

In the second case, for the test to pass, the last event must not be executed correctly: the `put` and `finish` are supposed to throw an exception if they are called; the `get` event is translated as in the case of trace events, but the check of the resulting value is inverted in this case. A test succeeds if one of the methods throws an exception or if the result of the *get* method corresponds to the incorrect value described in the test.

In order to check the resulting value returned by a call to the get method, an oracle function must be defined. As we already explained in section 2.2.3, using the equality functions provided by the implementation as an oracle is not safe. In our example, the equality method defined for the message type does not implement the desired behavior. Using this equality as an oracle will produce a lot of biased test results (and we have experienced that in our very first test campaign). For this reason, a specific equality function is defined and used to check the resulting values.

The three test cases presented previously, produce the following test methods:

```
1   public void testqueue1_1() throws Exception {
2       AbstractQueueableObjectImpl o_3_1 = new NamedEntry("topic", 3, 1);
3       AbstractQueueableObjectImpl o_0_0 = new NamedEntry("topic", 0, 0);
4       AbstractQueueableObjectImpl o_0_2 = new NamedEntry("topic", 0, 2);
5       AbstractQueueableObjectImpl o_0_1 = new NamedEntry("topic", 0, 1);
6       AbstractQueueableObjectImpl o_5_3 = new NamedEntry("topic", 5, 3);
7       AbstractQueueableObjectImpl o_1_4 = new NamedEntry("topic", 1, 4);
8       AbstractQueueableObjectImpl o = null;
9       tm.begin();
10
11      try {
12          queueManager.put(o_3_1); tm.commit();
13      } catch (Exception e) {
14          System.out.println("inconclusive"); return;
15      }
16
17      try {
18          queueManager.put(o_0_0); tm.commit();
19      } catch (Exception e) {
20          System.out.println("inconclusive"); return;
21      }
22
23      try {
24          queueManager.put(o_0_2); tm.commit();
25      } catch (Exception e) {
26          System.out.println("inconclusive"); return;
27      }
28
29      try {
30          queueManager.put(o_0_1); tm.commit();
31      } catch (Exception e) {
32          System.out.println("inconclusive"); return;
33      }
```

```
34
35          try {
36              queueManager.put(o_5_3); tm.commit();
37          } catch (Exception e) {
38              System.out.println("inconclusive"); return;
39          }
40
41          try {
42              queueManager.put(o_1_4); tm.commit();
43          } catch (Exception e) {
44              System.out.println("inconclusive"); return;
45          }
46
47          o = queueManager.get(NamedEntry.class, "topic");
48          assertFalse(equals(o_0_1, o));
49      }
```

```
1       public void testqueue1_2() throws Exception {
2           AbstractQueueableObjectImpl o_0_0 = new NamedEntry("topic", 0, 0);
3           AbstractQueueableObjectImpl o_3_1 = new NamedEntry("topic", 3, 1);
4           AbstractQueueableObjectImpl o_0_1 = new NamedEntry("topic", 0, 1);
5           AbstractQueueableObjectImpl o_3_2 = new NamedEntry("topic", 3, 2);
6           AbstractQueueableObjectImpl o = null;
7           tm.begin();
8
9           try {
10              queueManager.put(o_0_0); tm.commit();
11          } catch (Exception e) {
12              System.out.println("inconclusive"); return;
13          }
14
15          try {
16              queueManager.put(o_3_1); tm.commit();
17          } catch (Exception e) {
18              System.out.println("inconclusive"); return;
19          }
20
21          try {
22              queueManager.put(o_0_1); tm.commit();
23          } catch (Exception e) {
24              System.out.println("inconclusive"); return;
25          }
26
27          try {
28              queueManager.put(o_0_0); tm.commit();
29          } catch (Exception e) {
30              System.out.println("inconclusive"); return;
31          }
32
33          try {
34              queueManager.put(o_0_0); tm.commit();
35          } catch (Exception e) {
36              System.out.println("inconclusive"); return;
37          }
38
39          try {
40              queueManager.put(o_0_0); tm.commit();
```

```
41          } catch (Exception e) {
42              System.out.println("inconclusive"); return;
43          }
44
45          o = queueManager.get(NamedEntry.class, "topic");
46          assertFalse(equals(o_3_2, o));
47      }
```

```
1       public void testqueue1_3() throws Exception {
2           AbstractQueueableObjectImpl o_1_2 = new NamedEntry("topic", 1, 1);
3           AbstractQueueableObjectImpl o_1_6 = new NamedEntry("topic", 1, 6);
4           AbstractQueueableObjectImpl o_1_0 = new NamedEntry("topic", 1, 0);
5           AbstractQueueableObjectImpl o_2_0 = new NamedEntry("topic", 2, 0);
6           AbstractQueueableObjectImpl o_2_4 = new NamedEntry("topic", 2, 4);
7           AbstractQueueableObjectImpl o_0_4 = new NamedEntry("topic", 0, 4);
8           AbstractQueueableObjectImpl o_0_1 = new NamedEntry("topic", 0, 1);
9           AbstractQueueableObjectImpl o = null;
10          tm.begin();
11
12          try {
13              queueManager.put(o_1_2); tm.commit();
14          } catch (Exception e) {
15              System.out.println("inconclusive"); return;
16          }
17
18          try {
19              queueManager.put(o_1_6); tm.commit();
20          } catch (Exception e) {
21              System.out.println("inconclusive"); return;
22          }
23
24          try {
25              queueManager.put(o_1_0); tm.commit();
26          } catch (Exception e) {
27              System.out.println("inconclusive"); return;
28          }
29
30          try {
31              queueManager.put(o_2_0); tm.commit();
32          } catch (Exception e) {
33              System.out.println("inconclusive"); return;
34          }
35
36          try {
37              queueManager.put(o_2_4); tm.commit();
38          } catch (Exception e) {
39              System.out.println("inconclusive"); return;
40          }
41
42          try {
43              queueManager.put(o_0_4); tm.commit();
44          } catch (Exception e) {
45              System.out.println("inconclusive"); return;
46          }
47
48          try {
49              queueManager.finish(o_0_1); tm.commit();
```

```
50              fail("expected exception was not thrown");
51          } catch (Exception e) {
52              // expected
53          }
54      }
```

The exhaustive test generation produces more than 1000 test methods from all the traces of length smaller than 6. All resulting test methods are collected in a Java test file and executed against the implementation. This compilation is not automated. It is done using some external tool, in order to group all resulting test methods into one Java class. We could avoid this manual step by generating automatically a class for each test case containing its test methods. In our case, this will result in around 150 Java classes for each conformance relation. In the future, we plan to make the generation fully automatic, and thus all generated tests can be grouped into a single Java class.

The resulting Java class is integrated to the Eclipse environment in order to be executed. A screen-shot of the test execution environment is given in Figure 6.3. The execution of the 1087 generated test methods ended with no errors and 479 inconclusive tests. The analysis of these results is given in the next section.

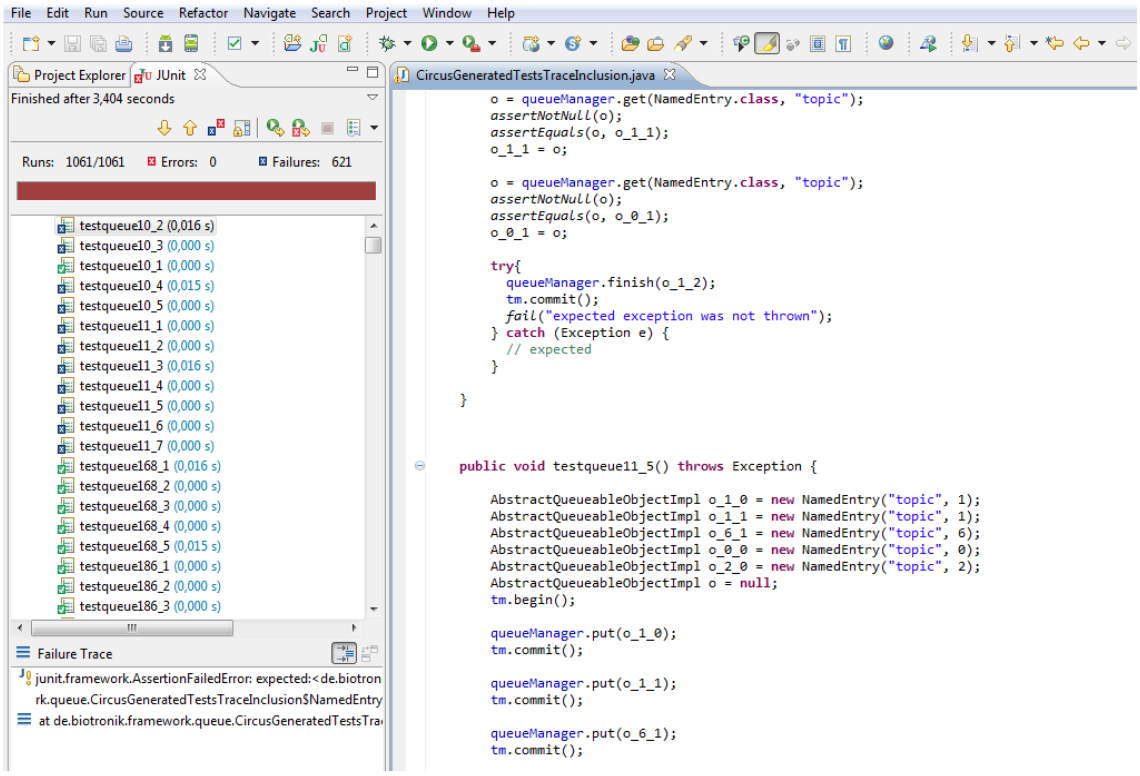



Figure 6.3: Test execution environment

#### 6.4.2.2 Deadlock reduction

The generated tests for the second conformance relation are also automatically translated into Java methods. This is done using a method called `export_test_file2`, which is slightly different from the first one. The translation rules are the same for the (sub)traces, by transforming each event into the corresponding method. For the acceptances set, the translation is more tricky. Since our acceptances sets are finite, the concrete acceptances can be enumerated and translated to produce the following behavior: First, the queue state is saved using a *commit* operation. Then for the first acceptance event, the corresponding method is called. If the call fails, the queue state is retrieved using the *rollback* operation and the execution continues with the remaining acceptances. As soon as a call is successfully performed, the test passes. If all the acceptances fail then the test fails as well.

In the special case of infinite acceptances sets (explicitly enumerating an unbounded number of inputs) the translation will be slightly different. The instantiation is not possible at the generation step, so the symbolic test must be translated directly to the corresponding method call. The obtained input is used to check if the constraint associated to this test is satisfied.

The concrete test case generated previously produces the following test method:

```
1   public void testqueue1_1() throws Exception {
2       AbstractQueueableObjectImpl o_2_0 = new NamedEntry("topic", 2, 0);
3       AbstractQueueableObjectImpl o_2_4 = new NamedEntry("topic", 2, 4);
4       AbstractQueueableObjectImpl o_0_4 = new NamedEntry("topic", 0, 4);
5       AbstractQueueableObjectImpl o_0_1 = new NamedEntry("topic", 0, 1);
6       AbstractQueueableObjectImpl o_0_0 = new NamedEntry("topic", 0, 0);
7       AbstractQueueableObjectImpl o_0_3 = new NamedEntry("topic", 0, 3);
8       AbstractQueueableObjectImpl o = null;
9       tm.begin();
10
11      try {
12          queueManager.put(o_2_0); tm.commit();
13      } catch (Exception e) {
14          System.out.println("inconclusive"); return;
15      }
16
17      try {
18          queueManager.put(o_2_4); tm.commit();
19      } catch (Exception e) {
20          System.out.println("inconclusive"); return;
21      }
22
23      try {
24          queueManager.put(o_0_4); tm.commit();
25      } catch (Exception e) {
26          System.out.println("inconclusive"); return;
27      }
28
29      try {
30          queueManager.put(o_0_1); tm.commit();
```

```
31          } catch (Exception e) {
32              System.out.println("inconclusive"); return;
33          }
34
35          try {
36              queueManager.put(o_2_0); tm.commit();
37          } catch (Exception e) {
38              System.out.println("inconclusive"); return;
39          }
40
41          try {
42              queueManager.put(o_0_0); tm.commit();
43          } catch (Exception e) {
44              System.out.println("inconclusive"); return;
45          }
46
47          try {
48              o = queueManager.get(NamedEntry.class, "topic"); tm.commit();
49          } catch (Exception e) {
50              tm.rollback(); queueManager.put(o_0_3); tm.commit();
51          }
52
53          if (o == null || !equals(o_2_0, o)) {
54              tm.rollback(); queueManager.put(o_0_3); tm.commit();
55          }
56      }
```

The exhaustive test generation produces not less than 660 test methods from all the traces of length smaller than 6. As for trace inclusion, all resulting test methods are collected in a Java test file and executed against the implementation. The execution of the 660 generated test methods revealed no errors and 315 inconclusive tests. These results are discussed in the next section.

### 6.4.3 Test results

The exhaustive test generation for length at most 7 produced a total of 1747 test cases using the 150 traces of length smaller or equal to 6. Using the generic rules, the generation time was very significant. This time was reduced by using some specific factorized rules, but it is still important (5 minutes for traces of length 7). All the generated test cases are concrete, where all communicated values are instantiated. As explained before, all these test cases are compiled into Java test methods that are executed using JUnit. The test execution time is negligible *w.r.t.* the test generation time (less than 10 seconds).

For the trace-inclusion conformance relation, the execution of the 1087 test methods ended without finding errors, but with 479 inconclusive tests. In the second case of deadlock reduction relation, no errors and 315 inconclusive tests resulted from executing the 660 test cases. The specification being very abstract at this stage, the generated tests cover a simple sequential behavior of the queue. This behavior was

intensively tested during the development of the system; no errors were detected by our generated tests.

Almost half of the test cases ended with an inconclusive verdict (794 from 1747), which reduces the efficiency of our test. In order to reduce the number of inconclusive tests, one possible solution is to combine the tests of the two conformance relations. The structure of the tests will be more complex (as a tree for example), but the number of resulting tests would be smaller.

## 6.5 Conclusions

This chapter presents an application of our specification and test-generation environment (*CirTA*) to a realistic case study. The studied system is a message monitoring module implemented as a FIFO multi-queue. This module is a part of a remote monitoring system that connects a variety of devices and especially patients pacemaker controllers. The overall system is explained in the first section, with some details on the queue module. The Java implementation of this queue is used to execute the generated tests.

A *Circus* specification of the queue is given in the second section of this chapter. The queue operations are represented by communication channels and the queue is handled by state variables. The representation of this specification in the syntax of Isabelle/*Circus* is also given in the second section.

The rest of this chapter covers the test generation, preparation and execution. First, the queue specification is used to generate all the traces up to a given length. For each generated trace, the sets of the corresponding tests are generated for the two conformance relations. Tests are then translated into Java methods in order to be tested against the queue implementation. Test execution is done using the JUnit testing tool in the Eclipse platform.

The first test campaigns raised two important issues related to test generation and execution. First, the choice of the simplification rules used in the test generation is very critical. Using a big number of simplification rules increases the generation time, especially when the proof goal is very complex. Moreover, some simplification rules may lead to oversimplifications of the assumptions and thus a loss of constraints. The reason is that the simplifier may refine the starting specification to produce the resulting subgoals. The set of simplification rules has been reduced to avoid these problems. Only useful and safe rules are used in the test generation. All the rules that can induce a loss of constraints are removed from the set of simplification rules. The final proof state should be equivalent to the starting test specification, and not just a refinement of the later. This property might be proved for each simplification rule used by the simplifier.

A second important issue is the definition of a specific oracle function for the

evaluation of test results. As mentioned earlier, the use of the equality method defined in the implementation for the message type gave biased results. A specific equality function, based on the equality of integers, is defined and used to check the resulting messages values.

The tests revealed no errors, which is not a big surprise given that the system under test is already in use. However, this case study presents a proof of technology of how our environment can be used for a real system. On the basis of this environment, it remains to introduce more realistic testing strategies than exhaustivity. It will be done by introducing stronger test hypotheses, thus some guidance of the test-generation tactics.

CHAPTER 7

# Conclusions and Future Work

## Summary of Contributions

This PhD thesis presents a semantics-based approach for specification, verification and testing. We take advantage of the formal framework provided by theorem provers, in our case Isabelle/HOL. Such formal frameworks offer machine-checked support for any logics-based development.

The *Circus* language, whose semantics is based on a relational calculus, is the object of our formal developments. This language provides a notation and some techniques to reason about designs and implementations of specifications describing complex data and reactive processes. Its rich and fully integrated semantics makes it a very good formalism for specifying real world systems. The semantic integration of this language in the Isabelle/HOL framework is the basis of our work. On top of this integration, we formalize several verification and testing techniques for *Circus*.

The work presented in this thesis brings several contributions to the field of formal specification, verification and testing environments. In itself, the formal semantics-based representation of a complex and rich specification language in a theorem prover is an important achievement.

The main results of this thesis are listed below:

**UTP in Isabelle/HOL** The first contribution is the shallow embedding of UTP, the semantics basis of *Circus*, in Isabelle/HOL. UTP stands for Unifying Theories of Programming, which is a semantic model based on an alphabetized relational calculus. UTP offers a set of theories that can be used to unify the semantics of

languages from different paradigms. It is used, for instance, to define the semantics of *Circus*, combining the notions of complex data manipulations and complex behavior descriptions. We introduce in section 4.2 a formalization of UTP in Isabelle/HOL. This formalization is expressed as a shallow embedding of the UTP relational calculus in HOL.

**Circus denotational semantics in Isabelle/HOL** The representation of the denotational semantics of *Circus* on top of UTP Isabelle encoding is another contribution of this work, with the definitions of several key notions like state variables, scoping, communication channels and name sets. These definitions make it possible to introduce a shallow embedding of the *Circus* language denotational semantics in Isabelle/HOL, given in section 4.3. This shallow embedding is based on the above formalization of UTP. Existing standard proof tactics of Isabelle such as `auto` and `simp` can be reused to reason about *Circus* specifications.

**Isabelle/Circus** We provide the Isabelle/*Circus* environment, based on the embedding of *Circus* in Isabelle/HOL. It can be used to reason about *Circus* specifications, or as a basis for a number of formal applications combining data and behavior descriptions. An interesting exercise is explored in section 4.4, with the formalization of one of the refinement notions of *Circus* with some support for refinement proofs.

**Circus operational semantics in Isabelle/HOL** In chapter 5, we introduce another substantial application based on the Isabelle/*Circus* environment. We provide a formalization of the rules of the operational semantics on top of Isabelle/*Circus* and explain some representation choices. Using these rules, we derive a number of other rules that are important for test generation.

**CirTA** We describe in chapter 5 the *CirTA* (*Circus* Testing Automation) test-generation framework. Test definitions are introduced and then automatic-generation tactics are defined as proof methods in Isabelle. The test definitions and generation tactics are defined for *Circus* processes, using the operational semantics. The choice of a shallow symbolic representation using Isabelle's symbolic facilities for test definitions and generation has turned out to be judicious and convenient since it avoids lots of heavy technicalities (in particular in the management of symbolic alphabets).

**Case study** In Chapter 6, the test generation environment (*CirTA*) is applied to a real case study. The studied system is a message monitoring module that binds together a variety of devices and especially patients pacemaker controllers. A *Circus* specification is written for this system and is used to generate tests up to a given length. A Java implementation of the monitoring system is tested using the

generated tests. Test execution is done using the JUnit testing tool in the Eclipse platform.

# Future Work

The work presented in this thesis is a significant advance in the instrumentation of complex real system specification and verification. However, it is clear that a number of interesting questions and future improvements must be addressed. Below, we sketch those that we plan to work through in the future.

**Denotational semantics** At the *Circus* denotational semantics level, some actions that are not considered yet must be defined (see section 3.4). Complex communications operations are not yet considered. In addition to this, some generalized operators (*e.g.* indexed choices) are not yet defined in the theory. Adding these actions will make our supported language easier to use.

Moreover, in the current version of the theories, no operations over processes are considered. Basically, they are similar as for actions, but with a special handling of local states (which may be challenging). These operations need to be defined to enable the writing of complex specifications in a more modular way.

**Isabelle/Circus** At the Isabelle/*Circus* level, parsing facilities must be improved to cover complex schema definitions and operations. External *Circus* specifications can be considered by integrating the environment with the existing CZT-parser for *Circus*. More proof support for refinement can be added, including some automatic refinement proof tactics.

**Operational semantics** For the operational semantics, an important achievement would be to prove the consistency of the rules *w.r.t.* the denotational semantics. This problem was already addressed in the scope of the *Circus* semantics [WCGF07], but no mechanized proofs are provided.

Another improvement would be to cover the additional *Circus* actions mentioned above for the denotational semantics: some new rules must be added to the operational semantics to describe the behavior of these actions.

**CirTA** At the *CirTA* level, several improvements are necessary. First, the whole test generation and preparation activities must be made more automated, in order to improve the whole time dedicated to test generation and compilation.

The integration of the test-generation environment in HOL-TestGen would be extremely fruitful for both environments. A first (weak) integration was already

presented in the case study. It covers the reuse of HOL-TestGen for test selection and instantiation.

Clearly, the efficiency of the generation procedures must be improved. The case study revealed that the test generation time is very important, mainly because of its exhaustivity. But it is also due to the number of simplification rules that are tried at each step of the generation. The size of the proof state plays also an important role in the efficiency of the generation. Several improvements are possible, either by adding more parallelization to the generation tactic or by lightening the set of involved simplification rules. The efficiency can also be improved using some lazy simplification techniques. This requires *freezing* some parts of the test specification at some points, to avoid the simplifier from attempting any simplifications on the frozen parts.

Last but not least, we have now the bases for more elaborated test selection, generation and execution strategies. A first step is some combination of the tests generated for the two conformance relations in order to reduce the number of inconclusive tests, with a proof that the exhaustivity results are kept. This requires the definition of new test structures and test-generation techniques. An other possible step is the formalization of coverage criteria, test purposes, and their associated selection hypotheses, based on the exploitation of both Isabelle/*Circus* and *CirTA*.

Moreover, on-line testing techniques can be used in order to reduce the number of inconclusive tests and of false negatives. This requires the testers to be connected with some constraint solvers to support on-line instantiations. We have already explored such techniques in a similar work covering web service testing [SWS].

APPENDIX A

# Circus syntax

## A.1 Full syntax

| | | |
|---|---|---|
| Program | ::= | CircusPar$^*$ |
| CircusPar | ::= | Par $\mid$ **channel** CDecl $\mid$ **chanset** N == CSExp $\mid$ ProcDecl |
| CDecl | ::= | SimpleCDecl $\mid$ SimpleCDecl; CDecl |
| SimpleCDecl | ::= | $N^+$ $\mid$ $N^+$ : Exp $\mid$ $[N^+]N^+$ : Exp $\mid$ SchemaExp |
| CSExp | ::= | $\{\!\mid \mid\!\}$ $\mid$ $\{\!\mid N^+ \mid\!\}$ $\mid$ N $\mid$ CSExp $\cup$ CSExp $\mid$ CSExp $\cap$ CSExp |
| | $\mid$ | CSExp $\setminus$ CSExp |
| ProcDecl | ::= | **process** N $\mathrel{\widehat=}$ ProcDef $\mid$ **process** $N[N^+]$ $\mathrel{\widehat=}$ ProcDef |
| ProcDef | ::= | Decl $\bullet$ ProcDef $\mid$ Decl $\odot$ ProcDef $\mid$ Proc |
| Proc | ::= | **begin** PPar$^*$ **state** SchemaExp PPar$^*$ $\bullet$ Action **end** |
| | $\mid$ | Proc; Proc $\mid$ Proc $\Box$ Proc $\mid$ Proc $\sqcap$ Proc |
| | $\mid$ | Proc $[\![$ CSExp $]\!]$ Proc $\mid$ Proc $\mid\mid\mid$ Proc $\mid$ Proc $\setminus$ CSExp |
| | $\mid$ | (Decl $\bullet$ ProcDef)$(Exp^+)$ $\mid$ $N(Exp^+)$ $\mid$ N |
| | $\mid$ | (Decl $\odot$ ProcDef)$\lfloor Exp^+ \rfloor$ $\mid$ $N\lfloor Exp^+ \rfloor$ $\mid$ Proc$[N^+ := N^+]$ |
| | $\mid$ | $N[Exp^+]$ $\mid$ $\fatsemi$ Decl $\bullet$ Proc $\mid$ $\Box$ Decl $\bullet$ Proc $\mid$ $\sqcap$ Decl $\bullet$ Proc |
| | $\mid$ | $[\![$CSExp$]\!]$ Decl $\bullet$ Proc $\mid$ $\mid\mid\mid$ Decl $\bullet$ Proc |
| NSExp | ::= | $\{\}$ $\mid$ $\{N^+\}$ $\mid$ N $\mid$ NSExp $\cup$ NSExp $\mid$ NSExp $\cap$ NSExp |
| | $\mid$ | NSExp $\setminus$ NSExp |
| PPar | ::= | Par $\mid$ N $\mathrel{\widehat=}$ ParAction $\mid$ **nameset** N == NSExp |
| ParAction | ::= | Action $\mid$ Decl $\bullet$ ParAction |
| Action | ::= | SchemaExp $\mid$ Command $\mid$ N $\mid$ CSPAction $\mid$ Action $[N^+ := N^+]$ |

| | | |
|---|---|---|
| CSPAction | ::= | $Skip$ \| $Stop$ \| $Chaos$ \| Comm $\rightarrow$ Action \| Pred $\&$ Action |
| | \| | Action; Action \| Action $\Box$ Action \| Action $\sqcap$ Action |
| | \| | Action $[\![$ NSExp \| CSExp \| NSExp $]\!]$ Action |
| | \| | Action $\|\|[$NSExp \| NSExp$]\|\|$ Action |
| | \| | Action $\setminus$ CSExp \| ParAction(Exp$^+$) \| $\mu$ N $\bullet$ Action |
| | \| | $\S$ Decl $\bullet$ Action \| $\Box$ Decl $\bullet$ Action \| $\sqcap$ Decl $\bullet$ Action |
| | \| | $[\![$CSExp $]\!]$ Decl $\bullet$ $[\![$NSExp $]\!]$ Action |
| | \| | $\|\|\|$ Decl $\bullet \|\|[$NSExp$]\|\|$ Action |
| Comm | ::= | N CParameter$^*$ \| N [Exp$^+$] CParameter$^*$ |
| CParameter | ::= | ?N \| ?N : Pred \| !Exp \| .Exp |
| Command | ::= | N$^+$ := Exp$^+$ \| **if** GActions **fi** \| **var** Decl $\bullet$ Action |
| | \| | N$^+$ : [ Pred, Pred ] \| {Pred} \| [Pred] |
| | \| | **val** Decl $\bullet$ Action \| **res** Decl $\bullet$ Action |
| | \| | **vres** Decl $\bullet$ Action |
| GActions | ::= | Pred $\rightarrow$ Action \| Pred $\rightarrow$ Action $\Box$ GActions |

## A.2 Short syntax

| | | |
|---|---|---|
| Program | ::= | CircusPar$^*$ |
| CircusPar | ::= | ZParagraph \| **channel** CDecl \| **chanset** N == CSExp |
| | \| | ProcDecl |
| CDecl | ::= | SimpleCDecl \| SimpleCDecl; CDecl |
| SimpleCDecl | ::= | N$^+$ \| N$^+$ : ZExp |
| CSExp | ::= | $\{\!\|$ $\|\!\}$ \| $\{\!\|$ N$^+$ $\|\!\}$ \| N \| CSExp $\cup$ CSExp \| CSExp $\cap$ CSExp |
| | \| | CSExp $\setminus$ CSExp |
| ProcDecl | ::= | **process** N $\widehat{=}$ ProcDef |
| ProcDef | ::= | **begin** PPar$^*$ **state** SchemaExp PPar$^*$ $\bullet$ ParAction **end** |
| NSExp | ::= | { } \| {N$^+$} \| N \| NSExp $\cup$ NSExp \| NSExp $\cap$ NSExp |
| | \| | NSExp $\setminus$ NSExp |
| PPar | ::= | ZParagraph \| N $\widehat{=}$ ParAction \| **nameset** N == NSExp |
| ParAction | ::= | CSPAction \| SchemaExp |
| CSPAction | ::= | $Skip$ \| $Stop$ \| $Chaos$ \| Command \| Comm $\rightarrow$ Action |
| | \| | Pred $\&$ Action \| $\mu$ N $\bullet$ Action \| Action $\setminus$ CSExp |
| | \| | Action; Action \| Action $\Box$ Action \| Action $\sqcap$ Action |
| | \| | Action $[\![$ NSExp \| CSExp \| NSExp $]\!]$ Action |
| Comm | ::= | N CParameter$^*$ |
| CParameter | ::= | ?N \| ?N : Pred \| !Exp \| .Exp |
| Command | ::= | N$^+$ := Exp$^+$ \| **var** Decl $\bullet$ Action |

## A.3 Isabelle/**Circus** syntax

| | | |
|---|---|---|
| Process | ::= | **circusprocess** Tpar* name = PParagraph* **where** Action |
| PParagraph | ::= | AlphabetP \| StateP \| ChannelP \| NamesetP \| ChansetP |
| | \| | ActionP \| SchemaP |
| AlphabetP | ::= | **alphabet** [ vardecl$^+$ ] |
| vardecl | ::= | name :: type |
| StateP | ::= | **state** [ vardecl$^+$ ] |
| ChannelP | ::= | **channel** [ chandecl$^+$ ] |
| chandecl | ::= | name \| name type |
| NamesetP | ::= | **nameset** name = [ name$^+$ ] |
| ChansetP | ::= | **chanset** name = [ name$^+$ ] |
| SchemaP | ::= | **schema** name = SchemaExpression |
| ActionP | ::= | **action** name = Action |
| Action | ::= | **Skip** \| **Stop** \| Action ; Action \| Action $\Box$ Action |
| | \| | Action $\sqcap$ Action \| Action $\setminus$ chansetN \| var := expr |
| | \| | guard $\&$ Action \| comm $\rightarrow$ Action \| **Schema** name |
| | \| | ActionName \| $\mu$ var $\bullet$ Action \| **var** var $\bullet$ Action |
| | \| | Action $[\![$ namesetN \| chansetN \| namesetN $]\!]$ Action |

APPENDIX B

# Circus denotational semantics

## B.1 UTP

### B.1.1 Observational variables

| | |
|---|---|
| $ok$ | Indicates if the system has been properly started in a stable state. |
| $ok'$ | Indicates if the system is in a subsequent stable observable state. |
| $wait'$ | Indicates if the subsequent state is an intermediate or a final state. |
| $tr$ | Records the trace of events performed before the system starts. |
| $tr'$ | Records the events performed by the system. |
| $ref$ | Collects the set of refused events before the system starts. |
| $ref'$ | Collects the set of refused events at a subsequent state. |
| $v$ | Indicates the initial values of program variables. |
| $v'$ | Indicates the final values of program variables. |

### B.1.2 Healthiness conditions

---

$H1$: A design may not make any prediction on variable values until the program has started.
`P = λ(A,A'). ok A →P (A,A')`

---

$H2$: A design may not require non-termination.
`P(A,A'⟦ok:=False⟧)→P(A,A'⟦ok:=True⟧)`

---

$H3$ : If the precondition of a design is satisfiable, its postcondition must be satisfiable too.
`P = P ;; Π`

---

$H4$ : Exclude miracle design.
`P ;; true = true`

---

$R1$ : The execution of a reactive process never undoes an event that has been performed.
`P = P ∧ λ (A,A'). tr A ≤ tr A'`

---

$R2$ : The behavior of a reactive process is oblivious to what has gone before.
`P = λ(A,A'). P(A⟦tr:=[]⟧,A'⟦tr:=(tr A' - tr A)⟧)`

---

$R3$ : Intermediate stable states do not progress.
`P = Πrea ◁ wait o fst ▷ P`

---

$CSP1$ : Extension of the trace is the only guarantee on divergence.
`P = P ∨ (λ (A,A'). ¬ ok A ∧ tr A ≤ tr A')`

---

$CSP2$ : A process may not require non-termination.
`P = P ;; J`
`J = λ(A,A'). (ok A →ok A') ∧ tr A' = tr A ∧wait A' = wait A`
`∧ ref A' = ref A ∧more A' = more A`

---

$C1$ : After termination, the value of $ref'$ has no relevance.
`P = P ; Skip`

---

$C2$ : A deadlocked process will stay deadlocked in an environment offering fewer events.
`P = P |[ ns1 | ns2 ]| Skip`

---

$C3$ : The precondition of a process expressed as a reactive design contains no dashed variables.
`P = R (¬P(A⟦wait:=false⟧,A'⟦ok:=false⟧); true ⊢`
`P(A⟦wait:=false⟧,A'⟦ok:=true⟧))`

---

where Π is the relational Skip, `o` is the HOL functional composition operator and `fst` returns the first element of a pair.

# B.2 Circus denotational semantics

## B.2.1 CSP actions

**Definition B.2.1.**

$$\begin{aligned} \mathit{II}_{rea} \mathrel{\widehat{=}} & (\neg\ okay \land tr \leq tr') \\ & \lor (okay' \land tr' = tr \land wait' = wait \land ref' = ref \land v' = v) \end{aligned}$$

**Definition B.2.2.** $Stop \mathrel{\widehat{=}} \boldsymbol{R}(true \vdash tr' = tr \land wait')$

**Definition B.2.3.** $Skip \mathrel{\widehat{=}} \boldsymbol{R}(true \vdash tr' = tr \land \neg\ wait' \land v' = v)$

**Definition B.2.4.** $Chaos \mathrel{\widehat{=}} \boldsymbol{R}(false \vdash true)$

**Definition B.2.5.** $A_1;\ A_2 \mathrel{\widehat{=}} A_1 ;_R A_2$

**Definition B.2.6.**

$$g\ \&\ A \mathrel{\widehat{=}} \boldsymbol{R}((g \rightarrow \neg\ A_f^f) \vdash ((g \land A_f^t) \lor (\neg\ g \land tr' = tr \land wait')))$$

**Definition B.2.7.**

$$\begin{aligned} & A_1 \Box A_2 \mathrel{\widehat{=}} \\ & \boldsymbol{R}((\neg\ {A_1}_f^f \land \neg\ {A_2}_f^f) \vdash (({A_1}_f^t \land {A_2}_f^t) \lhd tr' = tr \land wait' \rhd ({A_1}_f^t \lor {A_2}_f^t))) \end{aligned}$$

**Definition B.2.8.** $A_1 \sqcap A_2 \mathrel{\widehat{=}} A_1 \lor A_2$

**Definition B.2.9.**

$$do_{\mathcal{C}}\,(c, v) \mathrel{\widehat{=}} tr' = tr \land (c, v) \notin ref' \lhd wait' \rhd tr' = tr \frown \langle (c, v) \rangle$$

**Definition B.2.10.** $c \rightarrow Skip \mathrel{\widehat{=}} \boldsymbol{R}(true \vdash do_{\mathcal{C}}\,(c, Sync) \land v' = v)$

**Definition B.2.11.** $c.e \rightarrow Skip \mathrel{\widehat{=}} \boldsymbol{R}(true \vdash do_{\mathcal{C}}\,(c, e) \land v' = v)$

**Definition B.2.12.** $c!e \rightarrow Skip \mathrel{\widehat{=}} c.e \rightarrow Skip$

**Definition B.2.13.** *For any non-input communication* $c$*,*

$$c \rightarrow A \mathrel{\widehat{=}} (c \rightarrow Skip);\ A$$

**Definition B.2.14.**

$$
\begin{array}{l}
do_{\mathcal{I}}\,(c, x, P) \mathrel{\widehat=} \\
tr' = tr \land \{v : \delta(c) \mid P \bullet (c, v)\} \cap ref' = \emptyset \\
\lhd wait' \rhd \\
tr' - tr \in \{v : \delta(c) \mid P \bullet \langle (c, v) \rangle\} \land x' = snd(last(tr'))
\end{array}
$$

**Definition B.2.15.**

$$
c?x : P \;\rightarrow\; A(x) \mathrel{\widehat=} \; \mathbf{var}\; x \bullet \boldsymbol{R}(true \vdash do_{\mathcal{I}}\,(c, x, P) \land v' = v);\; A(x)
$$

**Definition B.2.16.** $c?x \;\rightarrow\; A \mathrel{\widehat=} \; c?x : true \;\rightarrow\; A$

**Definition B.2.17.**

$$
\begin{array}{l}
A \setminus cs \mathrel{\widehat=} \\
\boldsymbol{R}(\exists\, s \bullet A[s, cs \cup ref'/tr', ref'] \land (tr' - tr) = (s - tr) \upharpoonright (EVENT - cs)) \\
\qquad ;\; Skip
\end{array}
$$

**Definition B.2.18.** $\mu\, X \bullet F(X) \mathrel{\widehat=} \; \bigsqcap \{X \mid F(X) \sqsubseteq_{\mathcal{A}} X\}$

**Definition B.2.19.**

$$
A_1 \;[\![ns_2 \mid ns_2]\!]\; A_2 \mathrel{\widehat=}
$$

$$
\boldsymbol{R}\left(\begin{array}{l}
(\neg\, {A_1}_f^f \land \neg\, {A_2}_f^f) \\
\vdash \\
(({A_1}_f^t;\; U1(out\alpha\, A_1)) \land ({A_2}_f^t;\; U2(out\alpha\, A_2)))_{+\{v,tr\}};\; M_{|||_{cs}}
\end{array}\right)
$$

$$
\begin{array}{l}
M_{|||} \mathrel{\widehat=} \; tr' - tr \in (1.tr - tr \;|||\; 2.tr - tr) \\
\qquad\qquad \land \left(\begin{array}{l}
((1.wait \lor 2.wait) \land ref' \subseteq 1.ref \cap 2.ref) \\
\lhd wait' \rhd \\
(\neg\, 1.wait \land \neg\, 2.wait \land MSt)
\end{array}\right)
\end{array}
$$

$$
\begin{array}{lll}
\langle\rangle \;|||\; \langle\rangle \mathrel{\widehat=} & \{\langle\rangle\} & \\
tr_1 \;|||\; \langle\rangle \mathrel{\widehat=} & \{tr_1\} & \\
\langle\rangle \;|||\; tr_2 \mathrel{\widehat=} & \{tr_2\} & \\
e_1 : tr_1 \;|||\; e_2 : tr_2 \mathrel{\widehat=} & \{x \mid hd(x) = e_1 \land tl(x) \in (tr_1 \;|||\; e_2 : tr_2)\} & \\
& \cup & \\
& \{x \mid hd(x) = e_2 \land tl(x) \in (e_1 : tr_1 \;|||\; tr_2)\} &
\end{array}
$$

**Definition B.2.20.**

$$
\begin{array}{l}
A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 \mathrel{\widehat=} \\
\quad \boldsymbol{R} \left( \begin{array}{l}
\neg\ \exists 1.tr', 2.tr' \bullet (A_{1_f}^{\,f};\ 1.tr' = tr) \wedge (A_{2_f};\ 2.tr' = tr) \\
\qquad\qquad\qquad \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\
\wedge \neg\ \exists 1.tr', 2.tr' \bullet (A_{1_f};\ 1.tr' = tr) \wedge (A_{2_f}^{\,f};\ 2.tr' = tr) \\
\qquad\qquad\qquad \wedge\ 1.tr' \upharpoonright cs = 2.tr' \upharpoonright cs \\
\vdash \\
((A_{1_f}^{\,t};\ U1(out\alpha\, A_1)) \wedge (A_{2_f}^{\,t};\ U2(out\alpha\, A_2)))_{+\{v,tr\}};\ M_{\|_{cs}}
\end{array} \right) \\
U1(v_1', \ldots, v_n') \mathrel{\widehat=}\ 1.v_1' = v_1 \wedge \cdots \wedge 1.v_n' = v_n \\
\alpha U1(v_1', \ldots, v_n') = \{1.v_1', \ldots, 1.v_n', v_1, \ldots, v_n\} \\
U2(v_1', \ldots, v_n') \mathrel{\widehat=}\ 2.v_1' = v_1 \wedge \cdots \wedge 2.v_n' = v_n \\
\alpha U2(v_1', \ldots, v_n') = \{2.v_1', \ldots, 2.v_n', v_1, \ldots, v_n\} \\
MSt \mathrel{\widehat=}\ \forall v \bullet v \in ns_1 \rightarrow v' = 1.v \\
\qquad\qquad \wedge\ v \in ns_2 \rightarrow v' = 2.v \\
\qquad\qquad \wedge\ v \notin ns_1 \cup ns_2 \rightarrow v' = v \\
M_{\|_{cs}} \mathrel{\widehat=}\ tr' - tr \in (1.tr - tr \,\|_{cs}\, 2.tr - tr) \\
\qquad \wedge\ 1.tr \upharpoonright cs = 2.tr \upharpoonright cs \\
\qquad \wedge \left( \begin{array}{l}
\left( \begin{array}{l} (1.wait \vee 2.wait) \\ \wedge\ ref' \subseteq ((1.ref \cup 2.ref) \cap cs) \cup ((1.ref \cap 2.ref) \setminus cs) \end{array} \right) \\
\lhd wait' \rhd \\
(\neg\ 1.wait \wedge \neg\ 2.wait \wedge MSt)
\end{array} \right) \\
\langle\rangle \,\|_{cs}\, \langle\rangle \mathrel{\widehat=}\ \{\langle\rangle\} \\
e : tr \,\|_{cs}\, \langle\rangle \mathrel{\widehat=}\ \{\langle\rangle\} \lhd e \in cs \rhd \{x \mid hd(x) = e \wedge tl(x) \in (tr \,\|_{cs}\, \langle\rangle)\} \\
\langle\rangle \,\|_{cs}\, e : tr \mathrel{\widehat=}\ \{\langle\rangle\} \lhd e \in cs \rhd \{x \mid hd(x) = e \wedge tl(x) \in (\langle\rangle \,\|_{cs}\, tr)\} \\
e : tr_1 \,\|_{cs}\, e : tr_2 \mathrel{\widehat=} \\
\{x \mid hd(x) = e \wedge tl(x) \in (tr_1 \,\|_{cs}\, tr_2)\} \\
\lhd e \in cs \rhd \\
\left( \begin{array}{l}
\{x \mid hd(x) = e \wedge tl(x) \in (tr_1 \,\|_{cs}\, e : tr_2)\} \\
\cup \\
\{x \mid hd(x) = e \wedge tl(x) \in (e : tr_1 \,\|_{cs}\, tr_2)\}
\end{array} \right) \\
e_1 : tr_1 \,\|_{cs}\, e_2 : tr_2 \mathrel{\widehat=} \\
\left( \begin{array}{l}
\{\langle\rangle\} \\
\lhd e_2 \in cs \rhd \\
\{x \mid hd(x) = e_2 \wedge tl(x) \in (e_1 : tr_1 \,\|_{cs}\, tr_2)\}
\end{array} \right) \\
\lhd e_1 \in cs \rhd \\
\left( \begin{array}{l}
\{x \mid hd(x) = e_1 \wedge tl(x) \in (tr_1 \,\|_{cs}\, e_2 : tr_2)\} \\
\lhd e_2 \in cs \rhd \\
\left( \begin{array}{l}
\{x \mid hd(x) = e_1 \wedge tl(x) \in (tr_1 \,\|_{cs}\, e_2 : tr_2)\} \\
\cup \\
\{x \mid hd(x) = e_2 \wedge tl(x) \in (e_1 : tr_1 \,\|_{cs}\, tr_2)\}
\end{array} \right)
\end{array} \right)
\end{array}
$$

#### B.2.1.1 Iterated operators

**Definition B.2.21.** $\fatsemi\ x : \langle v_1, \ldots, v_n \rangle \bullet A(x) \mathrel{\widehat{=}} A(v_1) ; \ldots ; A(v_n)$

**Definition B.2.22.** $\Box\, x : T \bullet A(x) \mathrel{\widehat{=}} A(v_1) \Box \cdots \Box A(v_n)$

**Definition B.2.23.** $\sqcap\, x : T \bullet A(x) \mathrel{\widehat{=}} A(v_1) \sqcap \cdots \sqcap A(v_n)$

**Definition B.2.24.**

$$
\begin{array}{l}
[\![ cs ]\!]\ x : \{v_1, \ldots, v_n\} \bullet [\![ ns(x) ]\!]\ A(x) \mathrel{\widehat{=}} \\
A(v_1) \\
[\![ ns(v_1) \mid cs \mid \bigcup \{ x : \{v_2, \ldots, v_n\} \bullet ns(x) \} ]\!] \\
\left( \begin{array}{l} \ldots \left( \begin{array}{l} A(v_{n-1}) \\ [\![ ns(v_{n-1}) \mid cs \mid ns(v_n) ]\!] \\ A(v_n) \end{array} \right) \end{array} \right)
\end{array}
$$

**Definition B.2.25.**

$$
\begin{array}{l}
[\!|\!|\!| x : \{v_1, \ldots, v_n\} \bullet [\!|\![ ns(x) ]\!|\!]\ A(x) \mathrel{\widehat{=}} \\
A(v_1) \\
[\!|\![ ns(v_1) \mid \bigcup \{ x : \{v_2, \ldots, v_n\} \bullet ns(x) \} ]\!|\!] \\
\left( \begin{array}{l} \ldots \left( \begin{array}{l} A(v_{n-1}) \\ [\!|\![ ns(v_{n-1}) \mid ns(v_n) ]\!|\!] \\ A(v_n) \end{array} \right) \end{array} \right)
\end{array}
$$

### B.2.2 Action invocations, parametrized actions and renaming

In what follows, we consider the function action, given its name.

**Definition B.2.26.** $N \mathrel{\widehat{=}} \mathrm{B}(N)$

**Definition B.2.27.** $N(e) \mathrel{\widehat{=}} \mathrm{B}(N)(e)$

**Definition B.2.28.** $(x : T \bullet A)(e) \mathrel{\widehat{=}} A[e/x]$

**Definition B.2.29.**

$$
\begin{array}{l}
A[old_1, \ldots, old_n := new_1, \ldots, new_n] \\
\mathrel{\widehat{=}} \\
A[old_1, \ldots, old_n / new_1, \ldots, new_n]
\end{array}
$$

### B.2.3 Commands

**Definition B.2.30.**

$$x_1, \ldots, x_n := e_1, \ldots, e_n \mathrel{\widehat=} \\ \boldsymbol{R}(true \vdash tr' = tr \land \neg\ wait' \land x_1' = e_1 \land \cdots \land x_n' = e_n \land u' = u)$$

**Definition B.2.31.**

$$w : [\, pre, post \,] \mathrel{\widehat=} \boldsymbol{R}(pre \vdash post \land \neg\ wait' \land tr' = tr \land u' = u)$$

**Definition B.2.32.** $\{g\} \mathrel{\widehat=} : [g, true]$

**Definition B.2.33.** $[g] \mathrel{\widehat=} : [g]$

**Definition B.2.34.**

$$\textbf{if } [\!] \ i \bullet g_i A_i \textbf{ fi} \mathrel{\widehat=} \boldsymbol{R}((\bigvee i \bullet g_i) \land (\bigwedge i \bullet g_i \to \neg\ A_i{}^{f}_{f}) \vdash \bigvee i \bullet (g_i \land A_i{}^{t}_{f}))$$

**Definition B.2.35.** $\textbf{var } x : T \bullet A \mathrel{\widehat=} \textbf{var } x : T\textit{;}\ A\textit{;}\ \textbf{end } x : T$

**Definition B.2.36.**

$$(\textbf{val } x : T \bullet A)(e) \mathrel{\widehat=} (\textbf{var } x : T \bullet x := e\textit{;}\ A)$$

*provided* $x \notin FV(e)$

**Definition B.2.37.** $(\textbf{res } x : T \bullet A)(y) \mathrel{\widehat=} (\textbf{var } x : T \bullet A\textit{;}\ y := x)$

**Definition B.2.38.**

$$(\textbf{vres } x : T \bullet A)(y) \mathrel{\widehat=} (\textbf{var } x : T \bullet x := y\textit{;}\ A\textit{;}\ y := x)$$

*provided* $x \neq y$

### B.2.4 Schema expressions

**Definition B.2.39.** $[udecl;\ ddecl' \mid pred] \mathrel{\widehat=} ddecl : [\exists\, ddecl' \bullet pred, pred]$

### B.2.5 Circus processes

**Definition B.2.40.**

$$\textbf{begin state } [decl \mid pred]\ PPars\ \bullet A \textbf{ end} \mathrel{\widehat=} \textbf{var } decl \bullet A$$

**Definition B.2.41.** *For* $op \in \{ ; , \Box , \sqcap \}$*:*

$$\begin{array}{l} P \mathrel{op} Q \mathrel{\widehat{=}} \textbf{begin state } State \mathrel{\widehat{=}} P.State \land Q.State \\ \qquad\qquad P.PPar \land_{\Xi} Q.State \\ \qquad\qquad Q.PPar \land_{\Xi} P.State \\ \qquad\qquad \bullet\ P.Act \mathrel{op} Q.Act \\ \qquad \textbf{end} \end{array}$$

**Definition B.2.42.**

$$\begin{array}{l} P \mathrel{[\![ cs ]\!]} Q \mathrel{\widehat{=}} \textbf{begin state } State \mathrel{\widehat{=}} P.State \land Q.State \\ \qquad\qquad P.PPar \land_{\Xi} Q.State \\ \qquad\qquad Q.PPar \land_{\Xi} P.State \\ \qquad\qquad \bullet\ P.Act \mathrel{[\![} \alpha(P.State) \mid cs \mid \alpha(Q.State) \mathrel{]\!]} Q.Act \\ \qquad \textbf{end} \end{array}$$

**Definition B.2.43.**

$$\begin{array}{l} P \mathrel{|||} Q \mathrel{\widehat{=}} \textbf{begin state } State \mathrel{\widehat{=}} P.State \land Q.State \\ \qquad\qquad P.PPar \land_{\Xi} Q.State \\ \qquad\qquad Q.PPar \land_{\Xi} P.State \\ \qquad\qquad \bullet\ P.Act \mathrel{[\![} [\alpha(P.State) \mid \alpha(Q.State)] \mathrel{]\!]} Q.Act \\ \qquad \textbf{end} \end{array}$$

**Definition B.2.44.** $P \setminus cs \mathrel{\widehat{=}} \textbf{state } State \mathrel{\widehat{=}} P.State\ P.PPar\ \bullet P.Act \setminus cs \textbf{ end}$

**Definition B.2.45.** $x : T \odot P \mathrel{\widehat{=}} (x : T \bullet P)[c : usedC(P) \bullet c\_x.x]$

**Definition B.2.46.** $(x : T \odot P)\lfloor v \rfloor \mathrel{\widehat{=}} (x : T \odot P)(v)$

**Definition B.2.47.** $N\lfloor v_1 \rfloor \mathrel{\widehat{=}} \mathrm{B}(P)\lfloor v \rfloor$

**Definition B.2.48.** $N \mathrel{\widehat{=}} \mathrm{B}(N)$

**Definition B.2.49.** $N(e) \mathrel{\widehat{=}} \mathrm{B}(N)(e)$

**Definition B.2.50.** $(x : T \bullet P)(e) \mathrel{\widehat{=}} P[e/x]$

**Definition B.2.51.** $\fatsemi\ x : \langle v_1, \ldots, v_n \rangle \bullet P(x) \mathrel{\widehat{=}} P(v_1); \ \ldots ; \ P(v_n)$

**Definition B.2.52.** $\Box\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat{=}} P(v_1) \Box \cdots \Box P(v_n)$

**Definition B.2.53.** $\sqcap\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat{=}} P(v_1) \sqcap \cdots \sqcap P(v_n)$

**Definition B.2.54.**

$$[\![cs]\!]\ x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat{=}} P(v_1) \mathrel{[\![ cs ]\!]} (\ldots (P(v_{n-1}) \mathrel{[\![ cs ]\!]} P(v_n)))$$

**Definition B.2.55.** $\Vert\Vert\Vert\, x : \{v_1, \ldots, v_n\} \bullet P(x) \mathrel{\widehat{=}} P(v_1) \Vert\Vert\Vert (\ldots (P(v_{n-1}) \Vert\Vert\Vert P(v_n)))$

**Definition B.2.56.** $P[oldc := newc] \mathrel{\widehat{=}} P[newc/oldc]$

In what follows, we consider the function paragraphs and channels within a generic process (declared using generic parameters $T_0, \ldots, T_n$) with the types that are given.

**Definition B.2.57.** $P[te_0, \ldots, te_n] \mathrel{\widehat{=}} \mathrm{I}(\mathrm{B}(P), \langle te_0, \ldots, te_n \rangle)$

APPENDIX C

# Circus operational semantics

## C.1 Generic actions theorems

**Theorem 1.**

$$Schema\ sc = state\_update\_before\ sc\ Skip$$

**Theorem 2.**

$$v := e = state\_update\_before\ (\lambda(s, s').\ s' = (update\ v\ (\lambda\_.\ (e\ s)))\ s)\ Skip$$

**Theorem 3.**

$$\textbf{var}\ x \bullet A = state\_update\_before\ (\lambda(s, s').\ \exists\, a.\ s' = increase\ v\ a\ s)\ (\textbf{\textit{let}}\ v \bullet A)$$

**Theorem 4.**

$$\textbf{\textit{let}}\ x \bullet A = state\_update\_after\ (\lambda(s, s').\ s' = decrease\ v\ s)\ A$$

**Theorem 5.**

$$g\ \&\ P = state\_update\_before\ (\lambda(s, s').\ g\ s)\ P$$

**Theorem 6.**

$$c?x \in S \rightarrow P = iPrefix\ (\lambda\, s.\ c\ (select\ x\ s))\ (\lambda(s, s').\ \exists\, a.\ a \in S \land s' = increase\ x\ a\ s)\ (\lambda\, A.\ \textbf{\textit{let}}\ x \bullet A)\ (c`S)\ P$$

**Theorem 7.**

$$c!a \rightarrow P = oPrefix\ (\lambda\, s.\ c\ (a\ s))\ (range\ c)\ P$$

## C.2 Transition relation

### C.2.1 Introduction rules

$$\frac{c \qquad sc(s, s')}{(c \mid s \models state\_update\_before\ sc\ A) \xrightarrow{\varepsilon} (c \wedge sc(s, s') \mid s' \models A)} \tag{C.1}$$

$$\frac{c \qquad sc(s, s')}{(c \mid s \models state\_update\_after\ sc\ Skip) \xrightarrow{\varepsilon} (c \wedge sc(s, s') \mid s' \models Skip)} \tag{C.2}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad A1 \neq Skip}{(c_1 \mid s_1 \models state\_update\_after\ sc\ A_1) \xrightarrow{l} (c_2 \mid s_2 \models state\_update\_after\ sc\ A_2)} \tag{C.3}$$

$$\frac{c \qquad i(s, s')}{(c \mid s \models iPrefix\ d\ i\ j\ S\ Ac) \xrightarrow{in\ (d\ s')} (c \mid s' \models j\ Ac)} \tag{C.4}$$

$$\frac{c}{(c \mid s \models oPrefix\ d\ S\ Ac) \xrightarrow{out\ (d\ s)} (c \mid s \models Ac)} \tag{C.5}$$

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge (s;\ w_0 = e) \mid s;\ v := w_0 \models Skip)} \tag{C.6}$$

$$\frac{c}{(c \mid s \models v := e) \xrightarrow{\varepsilon} (c \wedge w_0 = e\ s \mid v\_update\ w_0\ s \models Skip)} \tag{+}$$

$$\frac{c \qquad (s;\ \text{pre}\ Op)}{(c \mid s \models Op) \xrightarrow{\varepsilon} (c \wedge (s;\ Op[w_0/v']) \mid s;\ v := w_0 \models Skip)} \quad v = out\alpha s \tag{C.7}$$

$$\frac{c \qquad Op(s, s')}{(c \mid s \models Op) \xrightarrow{\varepsilon} (c \wedge Op(s, s') \mid s' \models Skip)} \tag{+}$$

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models \mathbf{var}\ x : T \bullet A) \xrightarrow{\varepsilon} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \textbf{\textit{let}}\ x \bullet A)} \tag{C.8}$$

$$\frac{c}{(c \mid s \models \mathbf{var}\ x \bullet A) \xrightarrow{\varepsilon} (c \mid var\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet \mathit{Skip}) \xrightarrow{\varepsilon} (c \mid s;\ \mathbf{end}\ x \models \mathit{Skip})} \tag{C.9}$$

$$\frac{c}{(c \mid s \models \textbf{\textit{let}}\ x \bullet \mathit{Skip}) \xrightarrow{\varepsilon} (c \mid end\ x\ s \models \mathit{Skip})} \tag{+}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models \textbf{\textit{let}}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models \textbf{\textit{let}}\ x \bullet A_2)} \tag{C.10}$$

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{d!w_0} (c \wedge (s;\ w_0 = e) \mid s \models A)} \tag{C.11}$$

$$\frac{c}{(c \mid s \models d!e\ \rightarrow\ A) \xrightarrow{out\,(d\,w_0)} (c \wedge w_0 = e\,s \mid s \models A)} \tag{+}$$

$$\frac{c \wedge T \neq \emptyset \qquad x \notin \alpha s}{(c \mid s \models d?x : T\ \rightarrow\ A) \xrightarrow{d?w_0} (c \wedge w_0 \in T \mid s;\ \mathbf{var}\ x := w_0 \models \textbf{\textit{let}}\ x \bullet A)} \tag{C.12}$$

$$\frac{c}{(c \mid s \models d?x\ \rightarrow\ A) \xrightarrow{in\,(d\,w_0)} (c \mid var\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

$$\frac{c \qquad w_0 \in T}{(c \mid s \models d?x \in T\ \rightarrow\ A) \xrightarrow{in\,(d\,w_0)} (c \wedge w_0 \in T \mid var\ x\ w_0\ s \models \textbf{\textit{let}}\ x \bullet A)} \tag{+}$$

$$\frac{c}{(c \mid s \models d \ \rightarrow \ A) \xrightarrow{d} (c \mid s \models A)} \tag{C.13}$$

$$\frac{c}{(c \mid s \models d \ \rightarrow \ A) \xrightarrow{ev\,(d)} (c \mid s \models A)} \tag{+}$$

$$\frac{c}{(c \mid s \models \mathit{Skip} \,;\, A) \xrightarrow{\varepsilon} (c \mid s \models A)} \tag{C.14}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2)}{(c_1 \mid s_1 \models A_1 \,;\, B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \,;\, B)} \tag{C.15}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad A_1 \neq \mathit{Skip}}{(c_1 \mid s_1 \models A_1 \,;\, B) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \,;\, B)} \tag{+}$$

$$\frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_1)} \qquad \frac{c}{(c \mid s \models A_1 \sqcap A_2) \xrightarrow{\varepsilon} (c \mid s \models A_2)} \tag{C.16}$$

$$\frac{c}{(c \mid s \models A_1 \,\Box\, A_2) \xrightarrow{\varepsilon} (c \mid s \models (\mathsf{loc}\ c \mid s \bullet A_1) \boxplus (\mathsf{loc}\ c \mid s \bullet A_2))} \tag{C.17}$$

$$\frac{c_1}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet \mathit{Skip}) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A)) \xrightarrow{\varepsilon} (c_1 \mid s_1 \models \mathit{Skip})} \tag{C.18}$$

$$\frac{c_2}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet \mathit{Skip})) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models \mathit{Skip})} \tag{C.19}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{C.20}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{\varepsilon} (c_3 \mid s_3 \models A_3)}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{\varepsilon} \left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{C.21}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \varepsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)} \tag{C.22}$$

$$\frac{(c_2 \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l \neq \varepsilon}{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2)) \xrightarrow{l} (c_3 \mid s_3 \models A_3)} \tag{C.23}$$

$$\frac{c \wedge (s;\ g)}{(c \mid s \models g \,\&\, A) \xrightarrow{\varepsilon} (c \wedge (s;\ g) \mid s \models A)} \tag{C.24}$$

$$\frac{c \qquad (g\ s)}{(c \mid s \models g \,\&\, A) \xrightarrow{\varepsilon} (c \wedge g\ s \mid s \models A)} \tag{+}$$

$$\frac{c}{(c \mid s \models \mathit{Skip} \setminus cs) \xrightarrow{\varepsilon} (c \mid s \models \mathit{Skip})} \tag{C.25}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l \neq \varepsilon \quad \mathit{chan}\ l \notin cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \setminus cs)} \tag{C.26}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l \neq \varepsilon \quad \mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ l))\ es}{(c_1 \mid s_1 \models A_1 \setminus es) \xrightarrow{l} (c_2 \mid s_2 \models A_2 \setminus es)} \quad (+)$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l = \varepsilon \lor \mathit{chan}\ l \in cs}{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2 \setminus cs)} \quad \text{(C.27)}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \quad l = \varepsilon \lor \neg\, \mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ l))\ es}{(c_1 \mid s_1 \models A_1 \setminus es) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2 \setminus es)} \quad (+)$$

$$\frac{c}{(c \mid s \models A_1 \,[\![\, x_1 \mid cs \mid x_2 \,]\!]\, A_2) \xrightarrow{\varepsilon} \left( \begin{array}{l} c \mid s \\ \models \\ \quad \left( \begin{array}{c} (\mathsf{par}\ s \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s \mid x_2 \bullet A_2) \end{array} \right) \end{array} \right)} \quad \text{(C.28)}$$

$$\frac{c}{\left( \begin{array}{l} c \mid s \\ \models \\ \quad \left( \begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet \mathit{Skip}) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet \mathit{Skip}) \end{array} \right) \end{array} \right) \xrightarrow{\varepsilon} \left( \begin{array}{l} c \mid (\exists\, x_2' \bullet s_1) \land (\exists\, x_1' \bullet s_2) \\ \models \\ \quad \mathit{Skip} \end{array} \right)} \quad \text{(C.29)}$$

$$\frac{c}{\left( \begin{array}{l} c \mid s \\ \models \\ \quad \left( \begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet \mathit{Skip}) \\ [\![ns]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet \mathit{Skip}) \end{array} \right) \end{array} \right) \xrightarrow{\varepsilon} \left( \begin{array}{l} c \mid \mathit{StateMerge}\ (s_1, x_1)\ (s_2, x_2)\ s \\ \models \\ \quad \mathit{Skip} \end{array} \right)} \quad (+)$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee \mathit{chan}\, l \notin cs}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{C.30}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee (\mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ ll))\ es) \quad c_3}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket es \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ \llbracket es \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right)} \tag{+}$$

$$\frac{(c \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee \mathit{chan}\, l \notin cs}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_3 \mid x_2 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{C.31}$$

$$\frac{(c \mid s_2 \models A_2) \xrightarrow{l} (c_3 \mid s_3 \models A_3) \quad l = \varepsilon \vee (\mathit{filter\_chan\_set}\ (\mathit{the}\ (\mathit{chan}\ ll))\ es) \quad c_3}{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket es \rrbracket \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{l} \left(\begin{array}{l} c_3 \mid s \\ \models \\ \quad \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket es \rrbracket \\ (\mathsf{par}\ s_3 \mid x_2 \bullet A_3) \end{array}\right) \end{array}\right)} \tag{+}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{d} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{d} (c_4 \mid s_4 \models A_4) \quad d \in cs \quad c_3 \wedge c_4}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d} \left(\begin{array}{l} c_3 \wedge c_4 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{C.32}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{evn\ d} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{evn\ d} (c_4 \mid s_4 \models A_4) \quad \neg\,(\mathit{filter\_chan\_set}\ d\ es) \quad c_3 \wedge c_4}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![es]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{evn\ d} \left(\begin{array}{l} c_3 \wedge c_4 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![es]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{+}$$

$$\frac{(c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \quad d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{d?w_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{C.33}$$

$$\frac{\begin{array}{c}(c \mid s_1 \models A_1) \xrightarrow{in\ d_1} (c_3 \mid s_3 \models A_3) \quad (c \mid s_2 \models A_2) \xrightarrow{in\ d_2} (c_4 \mid s_4 \models A_4) \\ \neg\,(\mathit{filter\_chan\_set}\ d_1\ es) \quad \neg\,(\mathit{filter\_chan\_set}\ d_2\ es) \quad c_3 \wedge c_4 \wedge d_1 = d_2\end{array}}{\left(\begin{array}{l} c \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![es]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2) \end{array}\right) \end{array}\right) \xrightarrow{in\ d_2} \left(\begin{array}{l} c_3 \wedge c_4 \wedge d_1 = d_2 \mid s \\ \models \\ \left(\begin{array}{c} (\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![es]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4) \end{array}\right) \end{array}\right)} \tag{+}$$

$$\frac{\left(\begin{array}{l}(c \mid s_1 \models A_1) \xrightarrow{d?w_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{d?w_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{d!w_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{d!w_2} (c_4 \mid s_4 \models A_4)\end{array}\right) \quad d \in cs \quad c_3 \wedge c_4 \wedge w_1 = w_2}{\left(\begin{array}{l}c \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![cs]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2)\end{array}\right)\end{array}\right) \xrightarrow{d!w_2} \left(\begin{array}{l}c_3 \wedge c_4 \wedge w_1 = w_2 \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![cs]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4)\end{array}\right)\end{array}\right)} \tag{C.34}$$

$$\frac{\left(\begin{array}{l}(c \mid s_1 \models A_1) \xrightarrow{in\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{out\ d_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{out\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{in\ d_2} (c_4 \mid s_4 \models A_4) \\ \vee \\ (c \mid s_1 \models A_1) \xrightarrow{out\ d_1} (c_3 \mid s_3 \models A_3) \ \wedge \ (c \mid s_2 \models A_2) \xrightarrow{out\ d_2} (c_4 \mid s_4 \models A_4)\end{array}\right) \quad \neg\,(\mathit{filter\_chan\_set}\ d_1\ es) \quad \neg\,(\mathit{filter\_chan\_set}\ d_2\ es) \quad c_3 \wedge c_4 \wedge d_1 = d_2}{\left(\begin{array}{l}c \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ [\![es]\!] \\ (\mathsf{par}\ s_2 \mid x_2 \bullet A_2)\end{array}\right)\end{array}\right) \xrightarrow{out\ d_2} \left(\begin{array}{l}c_3 \wedge c_4 \wedge d_1 = d_2 \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{par}\ s_3 \mid x_1 \bullet A_3) \\ [\![es]\!] \\ (\mathsf{par}\ s_4 \mid x_2 \bullet A_4)\end{array}\right)\end{array}\right)} \tag{+}$$

$$\frac{c}{(c \mid s \models \mu\ X \bullet A, \delta) \xrightarrow{\varepsilon} (c \mid s \models A, \delta \oplus \{X \mapsto A\})} \tag{C.35}$$

$$\frac{c}{(c \mid s \models X, \delta) \xrightarrow{\varepsilon} (c \mid s \models \delta\ X, \delta)} \tag{C.36}$$

$$\frac{\mathit{monotonic}\ A}{(c \mid s \models (\mu\ X \bullet A(X))) = (c \mid s \models A(\mu\ X \bullet A(X)))} \tag{+}$$

### C.2.2 Derived elimination rules

$$\frac{(c_1 \mid s_1 \models \mathit{Skip}) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}[\mathit{false}]\\ \vdots \\ Q\end{array}}{Q} \tag{C.37}$$

$$\frac{(c_1 \mid s_1 \models v := e) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = (c_1 \wedge w_0 = e\ s_1) \\ \wedge\ s_2 = v\_update\ w_0\ s_1 \\ \wedge\ A_2 = \mathit{Skip} \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1}{Q} \tag{C.38}$$

$$\frac{(c_1 \mid s_1 \models \mathit{Op}) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = (c_1 \wedge \mathit{Op}(s_1, s_1')) \wedge s_2 = s_1' \\ \wedge\ A_2 = \mathit{Skip} \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1 \quad \mathit{Op}(s_1, s_1')}{Q} \tag{C.39}$$

$$\frac{(c_1 \mid s_1 \models \mathbf{var}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = c_1 \wedge s_2 = \mathit{var}\ x\ w_0\ s_1 \\ \wedge\ A_2 = \mathbf{let}\ x \bullet A_1 \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1}{Q} \tag{C.40}$$

$$\frac{(c_1 \mid s_1 \models \mathbf{let}\ x \bullet \mathit{Skip}) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = c_1 \wedge s_2 = \mathit{end}\ x\ s_1 \\ \wedge\ A_2 = \mathit{Skip} \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1}{Q} \tag{C.41}$$

$$\frac{(c_1 \mid s_1 \models \mathbf{let}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_3) \\ \wedge\ A_2 = \mathbf{let}\ x \bullet A_3 \end{array}\right] \\ \vdots \\ Q\end{array} \qquad A_1 \neq \mathit{Skip}}{Q} \tag{C.42}$$

$$\frac{(c_1 \mid s_1 \models d!e \ \rightarrow \ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = (c_1 \wedge w_0 = e\ s_1) \wedge s_2 = s_1 \\ \wedge\ A_2 = A_1 \wedge l = out\ (d\ w_0) \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.43}$$

$$\frac{(c_1 \mid s_1 \models d?x \ \rightarrow \ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \wedge s_2 = (var\ x\ w_0\ s_1) \wedge \\ A_2 = \textbf{\textit{let}}\ x \bullet A_1 \wedge l = in\ (d\ w_0) \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.44}$$

$$\frac{(c_1 \mid s_1 \models d?x \in T \ \rightarrow \ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = (c_1 \wedge w_0 \in T) \wedge s_2 = (var\ x\ w_0\ s_1) \\ \wedge\ A_2 = \textbf{\textit{let}}\ x \bullet A_1 \wedge l = in\ (d\ w_0) \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1 \quad w_0 \in T}{Q} \tag{C.45}$$

$$\frac{(c_1 \mid s_1 \models d \ \rightarrow \ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \wedge s_2 = s_1 \wedge \\ A_2 = A_1 \wedge l = ev\ d \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.46}$$

$$\frac{(c_1 \mid s_1 \models \mathit{Skip};\ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \wedge s_2 = s_1 \\ \wedge\ A_2 = A_1 \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.47}$$

$$\frac{(c_1 \mid s_1 \models A_1;\ B) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_3) \\ \wedge\ A_2 = A_3;\ B \end{array}\right] \\ \vdots \\ Q \end{array} \qquad A_1 \neq \mathit{Skip}}{Q} \tag{C.48}$$

$$\frac{(c_1 \mid s_1 \models A_1 \sqcap A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \land s_2 = s_1 \land l = \varepsilon \\ \land\, (A_2 = A_1 \lor A_2 = A_3) \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.49}$$

$$\frac{(c_1 \mid s_1 \models A_1 \,\Box\, A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \land s_2 = s_1 \land l = \varepsilon \land \\ A_2 = (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_1 \mid s_1 \bullet A_3) \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.50}$$

$$\frac{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet \mathit{Skip}) \boxplus (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3)) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_1 \land s_2 = s_1 \land \\ A_2 = \mathit{Skip} \land l = \varepsilon \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_1}{Q} \tag{C.51}$$

$$\frac{(c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_3 \mid s_3 \bullet \mathit{Skip})) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\begin{array}{c} c_2 = c_3 \land s_2 = s_3 \land \\ A_2 = \mathit{Skip} \land l = \varepsilon \end{array}\right] \\ \vdots \\ Q \end{array} \qquad c_3}{Q} \tag{C.52}$$

$$\frac{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \end{array}\right) \end{array}\right) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c} \left[\left(\begin{array}{c} (c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_4 \mid s_4 \models A_4) \\ \land\, c_2 = c \land s_2 = s \land l = \varepsilon \land \\ A_2 = \left(\begin{array}{l} (\mathsf{loc}\ c_4 \mid s_4 \bullet A_4) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3) \end{array}\right) \end{array}\right) \lor \left(\begin{array}{c} (c_3 \mid s_3 \models A_3) \xrightarrow{\varepsilon} (c_4 \mid s_4 \models A_4) \\ \land\, c_2 = c \land s_2 = s \land l = \varepsilon \land \\ A_2 = \left(\begin{array}{l} (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_4 \mid s_4 \bullet A_4) \end{array}\right) \end{array}\right)\right] \\ \vdots \\ Q \end{array}}{Q} \tag{C.53}$$

$$\frac{\begin{array}{c}\left(\begin{array}{c} c \mid s \\ \models \\ \left(\begin{array}{c}(\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \\ \boxplus \\ (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3)\end{array}\right)\end{array}\right) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \left[\begin{array}{c}(c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_4 \mid s_4 \models A_4) \\ \wedge\ c_2 = c_4 \wedge s_2 = s_4 \wedge l \neq \varepsilon \wedge \\ A_2 = A_4\end{array}\right) \vee \left(\begin{array}{c}(c_3 \mid s_3 \models A_3) \xrightarrow{l} (c_4 \mid s_4 \models A_4) \\ \wedge\ c_2 = c_4 \wedge s_2 = s_4 \wedge l \neq \varepsilon \wedge \\ A_2 = A_4\end{array}\right)\right] \\ \vdots \\ Q\end{array}}{Q} \tag{C.54}$$

$$\frac{(c_1 \mid s_1 \models g\&A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = (c_1 \wedge g\ s_1) \wedge s_2 = s_1 \\ \wedge\ A_2 = A_1 \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1 \quad (g\ s_1)}{Q} \tag{C.55}$$

$$\frac{(c_1 \mid s_1 \models \mathit{Skip} \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c} c_2 = c_1 \wedge s_2 = s_1 \wedge \\ A_2 = \mathit{Skip} \wedge l = \varepsilon \end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1}{Q} \tag{C.56}$$

$$\frac{(c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{array}{c}\left[\begin{array}{c}(c_1 \mid s_1 \models A_1) \xrightarrow{ll} (c_3 \mid s_3 \models A_3) \wedge \\ c_2 = c_1 \wedge s_2 = s_1 \wedge A_2 = (A_3 \setminus cs) \wedge \\ \left(\begin{array}{c}(ll \neq \varepsilon \wedge \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ ll)) \wedge l = ll) \\ \vee \\ (ll \neq \varepsilon \wedge \neg\, \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ ll)) \wedge l = \varepsilon) \\ \vee \\ (ll = \varepsilon \wedge l = \varepsilon)\end{array}\right)\end{array}\right] \\ \vdots \\ Q\end{array} \qquad c_1 \quad A_1 \neq \mathit{Skip}}{Q} \tag{C.57}$$

$$\frac{(c_1 \mid s_1 \models A_1 \llbracket x_1 \mid cs \mid x_3 \rrbracket A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} c_2 = c_1 \land s_2 = s_1 \land l = \varepsilon \land \\ A_2 = \begin{pmatrix} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_1 \mid x_3 \bullet A_3) \end{pmatrix} \end{bmatrix} \\ \vdots \\ Q \qquad c_1}{Q} \tag{C.58}$$

$$\frac{(c \mid s \models \begin{pmatrix} (\mathsf{par}\ s_1 \mid x_1 \bullet \mathit{Skip}) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_3 \mid x_3 \bullet \mathit{Skip}) \end{pmatrix}) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} s_2 = \mathit{StateMerge}\ (s_1, x_1)\ (s_3, x_3)\ s \\ \land\ c_2 = c \land l = \varepsilon \land A_2 = \mathit{Skip} \end{bmatrix} \\ \vdots \\ Q \qquad c}{Q} \tag{C.59}$$

$$\frac{\begin{pmatrix} c \mid s \\ \models \\ \begin{pmatrix} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_3 \mid x_3 \bullet A_3) \end{pmatrix} \end{pmatrix} \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} ((\mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ l))\ cs) \lor l = \varepsilon) \land c_2 = c_4 \land s_2 = s \land \\ \begin{pmatrix} (c \mid s_1 \models A_1) \xrightarrow{l} (c_4 \mid s_4 \models A_4) \land A_2 = ((\mathsf{par}\ s_4 \mid x_1 \bullet A_4) \llbracket cs \rrbracket (\mathsf{par}\ s_3 \mid x_3 \bullet A_3)) \\ \lor \\ (c \mid s_3 \models A_3) \xrightarrow{l} (c_4 \mid s_4 \models A_4) \land A_2 = ((\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \llbracket cs \rrbracket (\mathsf{par}\ s_4 \mid x_3 \bullet A_4)) \end{pmatrix} \end{bmatrix} \\ \vdots \\ Q \qquad c_4}{Q} \tag{C.60}$$

$$\frac{\begin{pmatrix} c \mid s \\ \models \\ \begin{pmatrix} (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_3 \mid x_3 \bullet A_3) \end{pmatrix} \end{pmatrix} \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} \neg\ (\mathit{filter\_chan\_set}\ d\ cs) \land l = \mathit{evn}\ d \\ \land\ c_2 = (c_4 \land c_5) \land s_2 = s \land \\ (c \mid s_1 \models A_1) \xrightarrow{l} (c_4 \mid s_4 \models A_4) \land \\ (c \mid s_3 \models A_3) \xrightarrow{l} (c_5 \mid s_5 \models A_5) \\ \land\ A_2 = \begin{pmatrix} (\mathsf{par}\ s_4 \mid x_1 \bullet A_4) \\ \llbracket cs \rrbracket \\ (\mathsf{par}\ s_5 \mid x_3 \bullet A_5) \end{pmatrix} \end{bmatrix} \\ \vdots \\ Q \qquad c_4 \land c_5}{Q} \tag{C.61}$$

$$\frac{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l}(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ s_3 \mid x_3 \bullet A_3)\end{array}\right) \end{array}\right) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} \neg\,(\mathit{filter\_chan\_set}\ d_1\ cs) \\ \neg\,(\mathit{filter\_chan\_set}\ d_2\ cs) \\ \wedge\ c_2 = (c_4 \wedge c_5 \wedge d_1 = d_2) \\ \wedge\ l = \mathit{in}\ d_1 \wedge s_2 = s \wedge \\ (c \mid s_1 \models A_1) \xrightarrow{\mathit{in}\ d_1} (c_4 \mid s_4 \models A_4) \wedge \\ (c \mid s_3 \models A_3) \xrightarrow{\mathit{in}\ d_2} (c_5 \mid s_5 \models A_5) \\ \wedge\ A_2 = \left(\begin{array}{c}(\mathsf{par}\ s_4 \mid x_1 \bullet A_4) \\ \|[cs]\| \\ (\mathsf{par}\ s_5 \mid x_3 \bullet A_5)\end{array}\right) \end{bmatrix} \quad \begin{array}{c}\vdots \\ Q\end{array} \qquad c_4 \wedge c_5 \wedge d_1 = d_2}{Q} \tag{C.62}$$

$$\frac{\left(\begin{array}{l} c \mid s \\ \models \\ \quad \left(\begin{array}{l}(\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \\ \|[cs]\| \\ (\mathsf{par}\ s_3 \mid x_3 \bullet A_3)\end{array}\right) \end{array}\right) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \qquad \begin{bmatrix} \neg\,(\mathit{filter\_chan\_set}\ d_1\ cs) \\ \neg\,(\mathit{filter\_chan\_set}\ d_2\ cs) \\ \wedge\ c_2 = (c_4 \wedge c_5 \wedge d_1 = d_2) \\ \wedge\ l = \mathit{out}\ d_1 \wedge s_2 = s \wedge \\ (c \mid s_1 \models A_1) \xrightarrow{ev_1} (c_4 \mid s_4 \models A_4) \wedge \\ (c \mid s_3 \models A_3) \xrightarrow{ev_2} (c_5 \mid s_5 \models A_5) \wedge \\ \left(\begin{array}{l}(ev_1 = \mathit{in}\ d_1 \wedge ev_2 = \mathit{out}\ d2) \vee \\ (ev_1 = \mathit{out}\ d_1 \wedge ev_2 = \mathit{in}\ d2) \vee \\ (ev_1 = \mathit{out}\ d_1 \wedge ev_2 = \mathit{out}\ d2)\end{array}\right) \\ \wedge\ A_2 = \left(\begin{array}{c}(\mathsf{par}\ s_4 \mid x_1 \bullet A_4) \\ \|[cs]\| \\ (\mathsf{par}\ s_5 \mid x_3 \bullet A_5)\end{array}\right) \end{bmatrix} \quad \begin{array}{c}\vdots \\ Q\end{array} \qquad c_4 \wedge c_5 \wedge d_1 = d_2}{Q} \tag{C.63}$$

## C.2.3 Other derived rules

$$\frac{}{\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models \mathit{Skip}) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\} = \{\}} \tag{C.64}$$

$$\frac{}{\begin{array}{c}\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models v := e) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \bigcup w_0 \in \{e\ s\} \bullet \{t \mid c_1 \wedge P\ t\ (c_1 \wedge w_0 = e\ s_1)\ (v\_update\ w_0\ s_1)\ \mathit{Skip}\ \varepsilon\}\end{array}} \tag{C.65}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models Op) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \bigcup s' \in \{s'' \mid Op(s_1, s'')\} \bullet \{t \mid c_1 \land P\ t\ (c_1 \land Op(s_1, s'))\ s'\ Skip\ \varepsilon\} \tag{C.66}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models \mathbf{var}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \bigcup w_0 \bullet \{t \mid c_1 \land P\ t\ c_1\ (var\ x\ w_0\ s_1)\ (\mathbf{let}\ x \bullet A)\ \varepsilon\} \tag{C.67}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models \mathbf{let}\ x \bullet Skip) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \{t \mid c_1 \land P\ t\ c_1\ (end\ x\ s_1)\ Skip\ \varepsilon\} \tag{C.68}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models \mathbf{let}\ x \bullet A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ (\mathbf{let}\ x \bullet A_2)\ l\} \tag{C.69}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models d!e\ \rightarrow\ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \bigcup w_0 \in \{e\ s_1\} \bullet \{t \mid c_1 \land P\ t\ (c_1 \land w_0 = e\ s_1)\ s_1\ A_1\ (out\ (d\ w_0))\} \tag{C.70}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models d?e\ \rightarrow\ A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \land P\ t\ c_2\ s_2\ A_2\ l\} \\ = \\ \bigcup w_0 \bullet \{t \mid c_1 \land P\ t\ c_1\ (var\ x\ w_0\ s_1)\ (\mathbf{let}\ x \bullet A_1)\ (in\ (d\ w_0))\} \tag{C.71}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models d?e \in T \to A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}$$
$$=$$
$$\bigcup w_0 \in T \bullet \{t \mid c_1 \wedge P\ t\ (c_1 \wedge w_0 \in T)\ (var\ x\ w_0\ s_1)\ (\textbf{\textit{let}}\ x \bullet A_1)\ (in\ (d\ w_0))\} \tag{C.72}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models d \to A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}$$
$$=$$
$$\{t \mid c_1 \wedge P\ t\ c_1\ s_1\ A_1\ (ev\ d)\} \tag{C.73}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1;\ B) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}$$
$$=$$
$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ (A_2;\ B)\ l\}$$
$$\cup$$
$$\{t \mid A_1 = \mathit{Skip} \wedge c_1 \wedge P\ t\ c_1\ s_1\ B\ \varepsilon\} \tag{C.74}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1 \sqcap A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}$$
$$=$$
$$\{t \mid c_1 \wedge P\ t\ c_1\ s_1\ A_1\ \varepsilon\} \quad \cup \quad \{t \mid c_1 \wedge P\ t\ c_1\ s_1\ A_3\ \varepsilon\} \tag{C.75}$$

$$\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1 \Box A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}$$
$$=$$
$$\{t \mid c_1 \wedge P\ t\ c_1\ s_1\ ((\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_1 \mid s_1 \bullet A_3))\ \varepsilon\} \tag{C.76}$$

$$
\begin{array}{l}
\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c \mid s \models (\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3)) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\} \\
= \\
\quad \{t \mid A_1 = \mathit{Skip} \wedge c_1 \wedge P\ t\ c_1\ s_1\ \mathit{Skip}\ \varepsilon\} \\
\quad \cup \\
\quad \{t \mid A_3 = \mathit{Skip} \wedge c_3 \wedge P\ t\ c_3\ s_3\ \mathit{Skip}\ \varepsilon\} \\
\quad \cup \\
\{t \mid \exists\, c_2\ s_2\ A_2 \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c\ s\ ((\mathsf{loc}\ c_2 \mid s_2 \bullet A_2) \boxplus (\mathsf{loc}\ c_3 \mid s_3 \bullet A_3))\ \varepsilon\} \\
\quad \cup \\
\{t \mid \exists\, c_2\ s_2\ A_2 \bullet (c_3 \mid s_3 \models A_3) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c\ s\ ((\mathsf{loc}\ c_1 \mid s_1 \bullet A_1) \boxplus (\mathsf{loc}\ c_2 \mid s_2 \bullet A_2))\ \varepsilon\} \\
\quad \cup \\
\quad \{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge l \neq \varepsilon \wedge P\ t\ c_2\ s_2\ A_2\ l\} \\
\quad \cup \\
\quad \{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_3 \mid s_3 \models A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge l \neq \varepsilon \wedge P\ t\ c_2\ s_2\ A_2\ l\}
\end{array}
\tag{C.77}
$$

$$
\begin{array}{c}
\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c \mid s \models (\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \llbracket cs \rrbracket (\mathsf{par}\ s_3 \mid x_3 \bullet A_3)) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\} \\
= \\
\{t \mid A_1 = \mathit{Skip} \wedge A_3 = \mathit{Skip} \wedge c \wedge P\ t\ c\ (\mathit{StateMerge}\ (s_1, x_1)\ (s_2, x_2)\ s)\ \mathit{Skip}\ \varepsilon\} \\
\cup \\
\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge c_2 \wedge \\
(l = \varepsilon \vee \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ l))\ cs) \wedge \\
P\ t\ c_2\ s\ ((\mathsf{par}\ s_2 \mid x_1 \bullet A_2) \llbracket cs \rrbracket (\mathsf{par}\ s_3 \mid x_3 \bullet A_3))\ l\} \\
\cup \\
\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c \mid s_3 \models A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge c_2 \wedge \\
(l = \varepsilon \vee \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ l))\ cs) \wedge \\
P\ t\ c_2\ s\ ((\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \llbracket cs \rrbracket (\mathsf{par}\ s_2 \mid x_3 \bullet A_2))\ l\} \\
\cup \\
\{t \mid \exists\, c_2\ s_2\ A_2\ d \bullet (c \mid s_3 \models A_3) \xrightarrow{\mathit{evn}\ d} (c_2 \mid s_2 \models A_2) \wedge (c \mid s_3 \models A_3) \xrightarrow{\mathit{evn}\ d} (c_4 \mid s_4 \models A_4) \wedge (c_2 \wedge c_4) \\
\wedge \neg\, \mathit{filter\_chan\_set}\ d\ cs \wedge P\ t\ (c_2 \wedge c_4)\ s\ ((\mathsf{par}\ s_2 \mid x_1 \bullet A_2) \llbracket cs \rrbracket (\mathsf{par}\ s_4 \mid x_3 \bullet A_4))\ (\mathit{evn}\ d)\} \\
\cup \\
\{t \mid \exists\, c_2\ s_2\ A_2\ d_1\ d_2 \bullet (c \mid s_3 \models A_3) \xrightarrow{\mathit{in}\ d_1} (c_2 \mid s_2 \models A_2) \wedge (c \mid s_3 \models A_3) \xrightarrow{\mathit{in}\ d_2} (c_4 \mid s_4 \models A_4) \\
\wedge (c_2 \wedge c_4 \wedge d_1 = d_2) \wedge \neg\, \mathit{filter\_chan\_set}\ d_1\ cs \wedge \neg\, \mathit{filter\_chan\_set}\ d_2\ cs \\
\wedge P\ t\ (c_2 \wedge c_4 \wedge d_1 = d_2)\ s\ ((\mathsf{par}\ s_2 \mid x_1 \bullet A_2) \llbracket cs \rrbracket (\mathsf{par}\ s_4 \mid x_3 \bullet A_4))\ (\mathit{in}\ d_1)\} \\
\cup \\
\{t \mid \exists\, c_2\ s_2\ A_2\ d_1\ d_2\ ev_1\ ev_2 \bullet (c \mid s_3 \models A_3) \xrightarrow{ev_1} (c_2 \mid s_2 \models A_2) \wedge (c \mid s_3 \models A_3) \xrightarrow{ev_2} (c_4 \mid s_4 \models A_4) \\
\wedge ((ev_1 = \mathit{in}\ d_1 \wedge ev_2 = \mathit{out}\ d2) \vee (ev_1 = \mathit{out}\ d_1 \wedge ev_2 = \mathit{in}\ d2) \vee (ev_1 = \mathit{out}\ d_1 \wedge ev_2 = \mathit{out}\ d2)) \\
\wedge (c_2 \wedge c_4 \wedge d_1 = d_2) \wedge \neg\, \mathit{filter\_chan\_set}\ d_1\ cs \wedge \neg\, \mathit{filter\_chan\_set}\ d_2\ cs \\
\wedge P\ t\ (c_2 \wedge c_4 \wedge d_1 = d_2)\ s\ ((\mathsf{par}\ s_2 \mid x_1 \bullet A_2) \llbracket cs \rrbracket (\mathsf{par}\ s_4 \mid x_3 \bullet A_4))\ (\mathit{out}\ d_1)\}
\end{array}
\tag{C.78}
$$

$$\frac{}{\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1 \llbracket x_1 \mid cs \mid x_3 \rrbracket A_3) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}}$$
$$= \quad \{t \mid c_1 \wedge P\ t\ c_1\ s_1\ ((\mathsf{par}\ s_1 \mid x_1 \bullet A_1) \llbracket\, cs \,\rrbracket (\mathsf{par}\ s_1 \mid x_3 \bullet A_3))\ \varepsilon\} \tag{C.79}$$

$$\frac{}{\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models g\ \&A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}}$$
$$= \quad \{t \mid c_1 \wedge (g\ s_1) \wedge P\ t\ (c_1 \wedge (g\ s_1))\ s_1\ A_1\ \varepsilon\} \tag{C.80}$$

$$\begin{array}{c}\frac{}{\{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1 \setminus cs) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge P\ t\ c_2\ s_2\ A_2\ l\}} \\ = \\ \{t \mid A_1 = \mathit{Skip} \wedge c_1 \wedge P\ t\ c_1\ s_1\ \mathit{Skip}\ \varepsilon\} \\ \cup \\ \{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge \\ \quad l \neq \varepsilon \wedge \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ l))\ cs \wedge P\ t\ c_2\ s_2\ (A_2 \setminus cs)\ l\} \\ \cup \\ \{t \mid \exists\, c_2\ s_2\ A_2\ l \bullet (c_1 \mid s_1 \models A_1) \xrightarrow{l} (c_2 \mid s_2 \models A_2) \wedge \\ \quad (l = \varepsilon \vee \neg\ \mathit{filter\_chan\_set}\ (\mathit{the}(\mathit{chan}\ l))\ cs) \wedge P\ t\ c_2\ s_2\ (A_2 \setminus cs)\ \varepsilon\}\end{array} \tag{C.81}$$

## C.3 Trace composition relation

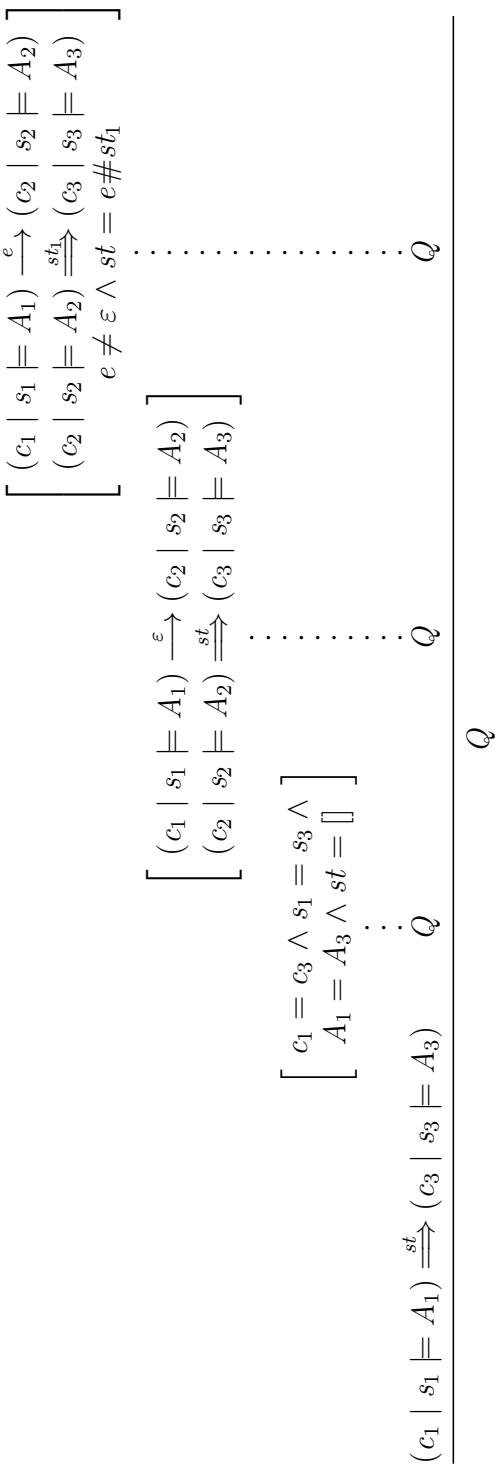
### C.3.1 Introduction rules

$$\frac{}{(c \mid s \models A) \overset{[]}{\Longrightarrow} (c \mid s \models A)} \tag{C.82}$$

$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2) \quad (c_2 \mid s_2 \models A_2) \overset{st}{\Longrightarrow} (c_3 \mid s_3 \models A_3)}{(c_1 \mid s_1 \models A_1) \overset{st}{\Longrightarrow} (c_3 \mid s_3 \models A_3)} \tag{C.83}$$

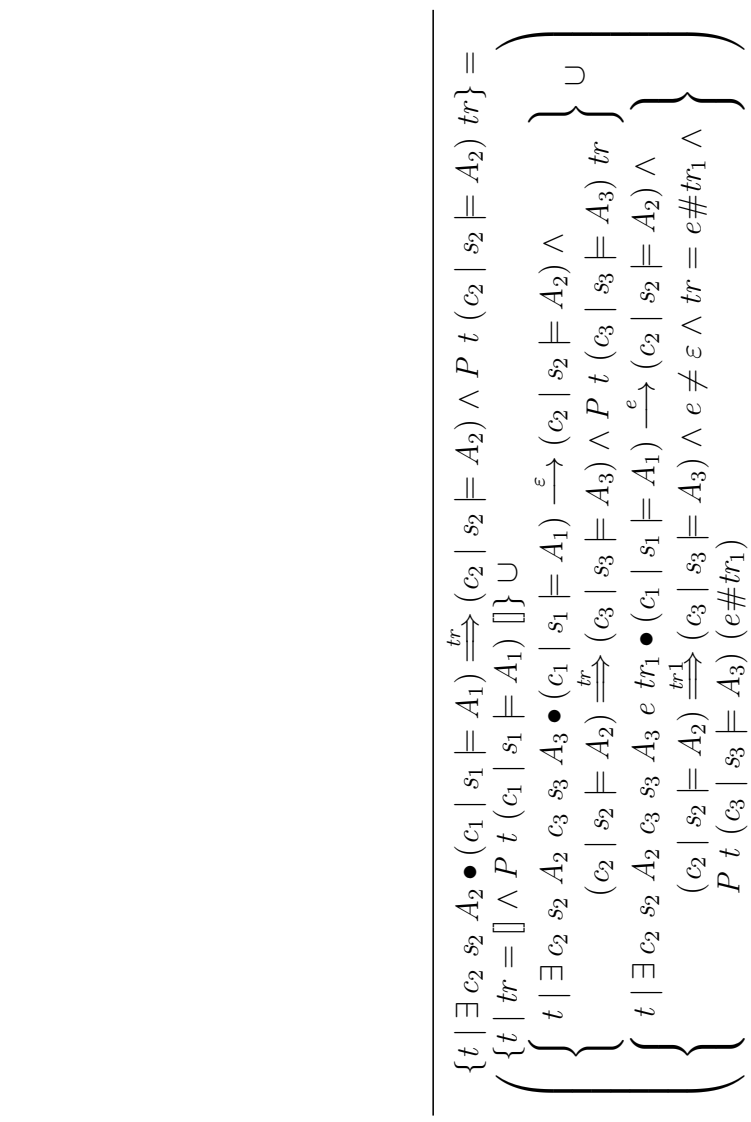
$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{e} (c_2 \mid s_2 \models A_2) \quad (c_2 \mid s_2 \models A_2) \overset{st}{\Longrightarrow} (c_3 \mid s_3 \models A_3) \quad e \neq \varepsilon}{(c_1 \mid s_1 \models A_1) \overset{e\#st}{\Longrightarrow} (c_3 \mid s_3 \models A_3)} \tag{C.84}$$

### C.3.2 Derived elimination rule



$$\frac{(c_1 \mid s_1 \models A_1) \xrightarrow{st} (c_3 \mid s_3 \models A_3) \qquad \begin{array}{l}\left[\begin{array}{l} c_1 = c_3 \wedge s_1 = s_3 \wedge \\ A_1 = A_3 \wedge st = [] \end{array}\right] \\ \vdots \\ Q\end{array} \qquad \begin{array}{l}\left[\begin{array}{l}(c_1 \mid s_1 \models A_1) \xrightarrow{\varepsilon} (c_2 \mid s_2 \models A_2) \\ (c_2 \mid s_2 \models A_2) \xRightarrow{st} (c_3 \mid s_3 \models A_3)\end{array}\right] \\ \cdots\cdots\cdots \\ Q\end{array} \qquad \begin{array}{l}\left[\begin{array}{l}(c_1 \mid s_1 \models A_1) \xrightarrow{e} (c_2 \mid s_2 \models A_2) \\ (c_2 \mid s_2 \models A_2) \xRightarrow{st_1} (c_3 \mid s_3 \models A_3) \\ e \neq \varepsilon \wedge st = e \# st_1\end{array}\right] \\ \cdots\cdots\cdots \\ Q\end{array}}{Q} \tag{C.85}$$

### C.3.3 Other derived rule



$$
\begin{array}{l}
\{t \mid \exists\, c_2\ s_2\ A_2 \bullet (c_1 \mid s_1 \models A_1) \overset{tr}{\Longrightarrow} (c_2 \mid s_2 \models A_2) \wedge P\ t\ (c_2 \mid s_2 \models A_2)\ tr\} = \\
\left(\begin{array}{l}
\{t \mid tr = [] \wedge P\ t\ (c_1 \mid s_1 \models A_1)\ []\} \cup \\
\left\{\begin{array}{l} t \mid \exists\, c_2\ s_2\ A_2\ c_3\ s_3\ A_3 \bullet (c_1 \mid s_1 \models A_1) \overset{\varepsilon}{\longrightarrow} (c_2 \mid s_2 \models A_2) \wedge \\ \qquad (c_2 \mid s_2 \models A_2) \overset{tr}{\Longrightarrow} (c_3 \mid s_3 \models A_3) \wedge P\ t\ (c_3 \mid s_3 \models A_3)\ tr \end{array}\right\} \cup \\
\left\{\begin{array}{l} t \mid \exists\, c_2\ s_2\ A_2\ c_3\ s_3\ A_3\ e\ tr_1 \bullet (c_1 \mid s_1 \models A_1) \overset{e}{\longrightarrow} (c_2 \mid s_2 \models A_2) \wedge \\ \qquad (c_2 \mid s_2 \models A_2) \overset{tr1}{\Longrightarrow} (c_3 \mid s_3 \models A_3) \wedge e \neq \varepsilon \wedge tr = e \# tr_1 \wedge \\ \qquad P\ t\ (c_3 \mid s_3 \models A_3)\ (e \# tr_1) \end{array}\right\}
\end{array}\right)
\end{array}
\tag{C.86}
$$

APPENDIX D

# Refinement laws

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P;\ P' \preceq_S Q;\ Q'}\ \text{SeqI} \tag{D.1}$$

$$\frac{P \preceq_S Q \qquad g_1 \simeq_S g_2}{g_1 \& P \preceq_S g_2 \& Q}\ \text{GrdI} \tag{D.2}$$

$$\frac{P \preceq_S Q \qquad x \sim_S y}{var\ x \bullet P \preceq_S var\ y \bullet Q}\ \text{VarI} \tag{D.3}$$

$$\frac{P \preceq_S Q \qquad x \sim_S y}{c?x \to P \preceq_S c?y \to Q}\ \text{InpI} \tag{D.4}$$

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \sqcap P' \preceq_S Q \sqcap Q'}\ \text{NdetI} \tag{D.5}$$

$$\frac{P \preceq_S Q \qquad x \sim_S y}{c!x \to P \preceq_S c!y \to Q}\ \text{OutI} \tag{D.6}$$

$$\frac{\begin{array}{c}[X \preceq_S Y] \\ \vdots \\ P\ X \preceq_S Q\ Y \end{array} \qquad mono\ P \qquad mono\ Q}{\mu\ X \bullet P\ X \preceq_S \mu\ Y \bullet Q\ Y}\ \text{MuI} \tag{D.7}$$

$$\frac{P \preceq_S Q \qquad P' \preceq_S Q'}{P \Box P' \preceq_S Q \Box Q'}\ \text{DetI} \tag{D.8}$$

$$\frac{\begin{array}{cc}[Pre\ sc_1\ (S\ A)] & [Pre\ sc_1\ (S\ A) \quad sc_2\ (A, A')] \\ \vdots & \vdots \\ Pre\ sc_2\ A & sc_1\ (S\ A, S\ A')\end{array}}{schema\ sc_1 \preceq_S schema\ sc_2}\ \text{SchI} \tag{D.9}$$

$$\frac{P \preceq_S Q}{a \rightarrow P \preceq_S a \rightarrow Q}\ \text{SyncI} \tag{D.10}$$

$$\frac{P \preceq_S Q \quad P' \preceq_S Q' \quad ns_1 \sim_S ns_1' \quad ns_2 \sim_S ns_2'}{P [\![ ns_1 \mid cs \mid ns_2 ]\!] P' \preceq_S Q [\![ ns_1' \mid cs \mid ns_2' ]\!] Q'}\ \text{ParI} \tag{D.11}$$

$$\frac{}{Skip \preceq_S Skip}\ \text{SkipI} \tag{D.12}$$

the relations "$x \sim_S y$" and "$g_1 \simeq_S g_2$" record the fact that the variable $x$ (repectively the guard $g_1$) is refined by the variable $y$ (repectively by the guard $g_2$) w.r.t the simulation function $S$.

# Bibliography


[AA92] N. Amla and P. Ammann. Using z specifications in category partition testing. In *Computer Assurance, 1992. COMPASS '92. 'Systems Integrity, Software Safety and Process Security: Building the System Right.', Proceedings of the Seventh Annual Conference on*, pages 3 –10, jun 1992. 19

[ABK+02] Egidio Astesiano, Michel Bidoit, Hélène Kirchner, Bernd Krieg-Brückner, Peter D. Mosses, Donald Sannella, and Andrzej Tarlecki. Casl: the common algebraic specification language. *Theor. Comput. Sci.*, 286(2):153–196, September 2002. 21

[Abr96] J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996. 20

[AO94] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 69 –79, jun-1 jul 1994. 19

[BB87] Tommaso Bolognesi and Ed Brinksma. Introduction to the iso specification language lotos. *Computer Networks*, 14:25–59, 1987. 26, 27

[BBC+10] Thomas Ball, Sebastian Burckhardt, Katherine E. Coons, Madanlal Musuvathi, and Shaz Qadeer. Preemption sealing for efficient concurrency testing. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 420–434, Berlin, Heidelberg, 2010. Springer-Verlag. 57

[BBN11] Jasmin Blanchette, Lukas Bulwahn, and Tobias Nipkow. Automatic proof and disproof in isabelle/hol. In Cesare Tinelli and Viorica Sofronie-Stokkermans, editors, *Frontiers of Combining Systems*, volume 6989 of

*Lecture Notes in Computer Science*, pages 12–27. Springer Berlin / Heidelberg, 2011. 32

[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, November 1986. 21

[BCM00] Simon Burton, John Clark, and John Mcdermid. Testing, proof and automation: An integrated approach. In *In Proc. 1st International Workshop of Automated Program Analysis, Testing and Verification*, 2000. 32

[BGG+92] Richard J. Boulton, Andrew Gordon, Michael J. C. Gordon, John Harrison, John Herbert, and John Van Tassel. Experience with embedding hardware description languages in hol. In *Proceedings of the IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, pages 129–156, Amsterdam, The Netherlands, The Netherlands, 1992. North-Holland Publishing Co. 63

[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, November 1991. 14, 15, 21

[BH04] Kirill Bogdanov and Mike Holcombe. Refinement in statechart testing. *Softw. Test., Verif. Reliab.*, 14(3):189–211, 2004. 27

[Bri88] Ed Brinksma. A theory for the derivation of tests. In S. Aggarwal and K. Sabnani, editors, *Protocol Specification, Testing, and Verification VIII*, volume 3582 of *Lecture Notes in Computer Science*, pages 63–74, North-Holland, 1988. Springer. 24

[BRW03] Achim D. Brucker, Frank Rittinger, and Burkhart Wolff. Hol-z 2.0: A proof environment for z-specifications. *Journal of Universal Computer Science*, 9(2):152–172, February 2003. 55, 63, 66, 68

[BT01] Ed Brinksma and Jan Tretmans. Testing transition systems: An annotated bibliography. In Franck Cassez, Claude Jard, Brigitte Rozoy, and Mark Ryan, editors, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 187–195. Springer Berlin / Heidelberg, 2001. 24

[BW07] Achim D. Brucker and Burkhart Wolff. Test-sequence generation with hol-testgen with an application to firewall testing. In *Proceedings of the*

*1st international conference on Tests and proofs*, TAP'07, pages 149–168, Berlin, Heidelberg, 2007. Springer-Verlag. 135

[BW08] Achim D. Brucker and Burkhart Wolff. An extensible encoding of object-oriented data models in hol with an application to imp++. *Journal of Automated Reasoning (JAR)*, 41(3–4):219–249, 2008. Serge Autexier, Heiko Mantel, Stephan Merz, and Tobias Nipkow (eds). 66

[BW12] Achim Brucker and Burkhart Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, pages 1–39, 2012. 10.1007/s00165-012-0222-y. 22, 32, 56, 96, 135, 155

[CG07] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in csp. In Michael Butler, Michael G. Hinchey, and María M. Larrondo-Petrie, editors, *ICFEM*, volume 4789 of *Lecture Notes in Computer Science*, pages 151–170. Springer, 2007. 22, 49

[CG10] Ana Cavalcanti and Marie-Claude Gaudel. Specification coverage for testing in circus. In *Proceedings of the Third international conference on Unifying theories of programming*, UTP'10, pages 1–45, Berlin, Heidelberg, 2010. Springer-Verlag. 116

[CG11] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in circus. *Acta Inf.*, 48(2):97–147, April 2011. 43, 47, 49, 98, 99, 101, 105, 108, 109, 116, 123, 127, 129, 132, 134, 140

[Cho78] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, May 1978. 14, 23

[Chu40] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):pp. 56–68, 1940. 51

[CM09] Maximiliano Cristiá and Pablo Rodríguez Monetti. Implementing and applying the stocks-carrington framework for model-based testing. In *Proceedings of the 11th International Conference on Formal Engineering Methods: Formal Methods and Software Engineering*, ICFEM '09, pages 167–185, Berlin, Heidelberg, 2009. Springer-Verlag. 20

[Coq] Coq web-site. http://coq.inria.fr/. 50

[CS94] David A. Carrington and Phil Stocks. A tale of two paradigms: Formal methods and software testing. In *Z User Workshop'94*, pages 51–68, 1994. 17, 20

[CSW03] Ana Cavalcanti, Augusto Sampaio, and Jim Woodcock. A refinement strategy for circus. *Formal Aspects of Computing*, 15:146–181, 2003. 90

[CW06] A. L. C. Cavalcanti and J. C. P. Woodcock. A tutorial introduction to csp in unifying theories of programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *Lecture Notes in Computer Science*, pages 220 – 268. Springer-Verlag, 2006. 71, 72

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the First International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, FME '93, pages 268–284, London, UK, UK, 1993. Springer-Verlag. 18, 19

[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. *Journal of Systems and Software*, 21(3):229–244, 1993. 21

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. 39

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. 56

[dNH83] R. de Nicola and M. Hennessy. Testing equivalences for processes. In Josep Diaz, editor, *Automata, Languages and Programming*, volume 154 of *Lecture Notes in Computer Science*, pages 548–560. Springer Berlin / Heidelberg, 1983. 10.1007/BFb0036936. 13, 22

[EFH83] H. Ehrig, W. Fey, and H. Hansen. ACT ONE: An algebraic specification language with two levels of semantics. Technical Report 83-01, Technische Universität Berlin, 1983. 21

[FGW10] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Unifying Theories in Isabelle/HOL. In *Unifying Theories of Programming (UTP2010)*, number 6445 in Lecture Notes in Computer Science, pages 188–206. Springer-Verlag, Heidelberg, November 2010. 20 pages. 62

[FGW11] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/*Circus* : a process specification and verification environment.

Technical Report 1547, LRI, Université Paris-Sud XI, November 2011. 62

[FGW12] Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/*Circus* : A process specification and verification environment. In Rajeev Joshi, Peter Müller, and Andreas Podelski, editors, *Verified Software: Theories, Tools, Experiments*, volume 7152 of *Lecture Notes in Computer Science*, pages 243–260. Springer Berlin / Heidelberg, 2012. 62

[Fis96] C. Fischer. Combining csp and z. Technical report, University of Oldenburg, 1996. 27

[Fis97] Clemens Fischer. Csp-oz: a combination of object-z and csp. In *Proceedings of the IFIP TC6 WG6.1 international workshop on Formal methods for open object-based distributed systems*, FMOODS '97, pages 423–438, London, UK, UK, 1997. Chapman & Hall, Ltd. 27

[For05] Formal. Formal systems (europe) ltd. failures-divergence refinement: Fdr2 user manual. available at http://www.fsel.com/fdr2_manual.html, June 2005. 22

[FTW06] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A symbolic framework for model-based testing. In *Proceedings of the First combined international conference on Formal Approaches to Software Testing and Runtime Verification*, FATES'06/RV'06, pages 40–54, Berlin, Heidelberg, 2006. Springer-Verlag. 25

[FWG12] Abderrahmane Feliachi, Burkhart Wolff, and Marie-Claude Gaudel. Isabelle/circus. *Archive of Formal Proofs*, May 2012. http://afp.sourceforge.net/entries/Circus.shtml, Formal proof development. 86

[Gal96] A. Galloway. *Integrated formal Methods with Richer Methodological Profiles for the Development of Multi-Perspective Systems.* PhD thesis, School of Computing and Mathematics, 1996. 27

[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*, TAPSOFT '95, pages 82–96, London, UK, UK, 1995. Springer-Verlag. 14, 15

[Gau01] Marie-Claude Gaudel. Testing from formal specifications, a generic approach. In Dirk Craeynest and Alfred Strohmeier, editors, *Reliable SoftwareTechnologies - Ada-Europe 2001*, volume 2043 of *Lecture Notes in Computer Science*, pages 35–48. Springer Berlin / Heidelberg, 2001. 22

[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. 15, 17

[Gau11] Marie-Claude Gaudel. Checking models, proving programs, and testing systems. In Martin Gogolla and Burkhart Wolff, editors, *TAP 2011 proceedings*, volume 6706 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2011. 13

[GJ98] M.-C. Gaudel and P. R. James. Testing algebraic data types and processes: a unifying theory. *Formal Aspects of Computing*, 10(5-6):436–451, 1998. 15

[GJK99] R. Groz, T. Jéron, and A. Kerbrat. Automated test generation from SDL specifications. In R. Dssouli, G. von Bochmann, and Y. Lahav, editors, *SDL'99 The Next Millenium, 9th SDL Forum, Montréal, Québec*, pages 135–152. Elsevier, June 1999. 26

[GLG08] Marie-Claude Gaudel and Pascale Le Gall. Formal methods and testing. chapter Testing data types implementations from algebraic specifications, pages 209–239. Springer-Verlag, Berlin, Heidelberg, 2008. 21, 56, 134

[GLMS11] Hubert Garavel, Frédéric Lang, Radu Mateescu, and Wendelin Serwe. Cadp 2010: A toolbox for the construction and analysis of distributed processes. In Parosh Abdulla and K. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 6605 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, 2011. 26

[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. 21

[GR98] Roland Groz and Nathalie Risser. Eight years of experience in test generation from fdts using tveda. In *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII)*, FORTE X / PSTV XVII '97, pages 465–480, London, UK, UK, 1998. Chapman & Hall, Ltd. 27

[GS97] A. J. Galloway and W. J. Stoddart. An operational semantics for zccs. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 272–, Washington, DC, USA, 1997. IEEE Computer Society. 27

[GT79] J.A. Goguen and J.J. Tardo. An introduction to obj: A language forwriting and testing formal algebraic program specifications. In *Proceedings of the Conference on Specificationsof Reliable Software*, pages 170–189, Boston, MA, 1979. 21

[Hal88] P.A.V. Hall. Towards testing with respect to formal specification. In *Software Engineering, 1988 Software Engineering 88., Second IEE/BCS Conference:*, pages 159 –163, jul 1988. 19

[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. 17

[HBH08] Robert M. Hierons, Jonathan P. Bowen, and Mark Harman, editors. *Formal Methods and Testing, An Outcome of the FORTEST Network, Revised Selected Papers*, volume 4949 of *Lecture Notes in Computer Science*. Springer, 2008. 57

[Hen64] F. C. Hennie. 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. 23

[HG96] David Harel and Eran Gery. Executable object modeling with statecharts. In *Proceedings of the 18th international conference on Software engineering*, ICSE '96, pages 246–257, Washington, DC, USA, 1996. IEEE Computer Society. 27

[HH98] C.A.R. Hoare and J. He. *Unifying theories of programming*, volume 14. Prentice Hall, 1998. 36, 40, 62

[Hie97] Robert M. Hierons. Testing from a z specification. *Softw. Test., Verif. Reliab.*, 7(1):19–33, 1997. 20

[Hie02] R. M. Hierons. Comparing test sets and criteria in the presence of test hypotheses and fault domains. *ACM Trans. Softw. Eng. Methodol.*, 11(4):427–448, October 2002. 14, 15

[Hie09] Robert M. Hierons. Verdict functions in testing with a fault domain or test hypotheses. *ACM Trans. Softw. Eng. Methodol.*, 18(4):14:1–14:19, July 2009. 14

[HNS97] Steffen Helke, Thomas Neustupny, and Thomas Santen. Automating test case generation from z specifications with isabelle. In *Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, ZUM '97, pages 52–71, London, UK, UK, 1997. Springer-Verlag. 20, 28, 32

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. 22

[HOL] Hol web-site. http://hol.sourceforge.net/. 50

[HSS01] Robert M. Hierons, Sadegh Sadeghipour, and Harbhajan Singh. Testing a system specified using statecharts and z. *Information & Software Technology*, 43(2):137–149, 2001. 27

[IT99] ITU-T. Recommendation z.100-specification and description language (sdl), 1999. 26

[JD07] Éric Jaeger and Catherine Dubois. Why would you trust b? In *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, LPAR'07, pages 288–302, Berlin, Heidelberg, 2007. Springer-Verlag. 63

[JJ04] C. Jard and T. Jéron. TGV: theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *Software Tools for Technology Transfer (STTT)*, 6, October 2004. 24, 25, 27, 57

[Jon86] Clifford B. Jones. *Systematic software development using VDM*. Prentice Hall International Series in Computer Science. Prentice Hall, 1986. 18

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. 31

[LB03] Michael Leuschel and Michael J. Butler. Prob: A model checker for b. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003. 27

[LC06] Yu Lei and Richard H. Carver. Reachability testing of concurrent programs. *IEEE Trans. Softw. Eng.*, 32(6):382–403, June 2006. 57

[LG02] Grégory Lestiennes and Marie-Claude Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, ISSRE '02, pages 3–, Washington, DC, USA, 2002. IEEE Computer Society. 15, 24, 25

[lot89] Iso 8807:1989 information processing systems – open systems interconnection – lotos – a formal description technique based on the temporal ordering of observational behaviour, 1989. 26, 27

[LPU02] Bruno Legeard, Fabien Peureux, and Mark Utting. Automated boundary testing from z and b. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, pages 21–40, London, UK, UK, 2002. Springer-Verlag. 20

[LY94] D. Lee and M. Yannakakis. Testing finite-state machines: State identification and verification. *IEEE Trans. Comput.*, 43(3):306–320, March 1994. 14, 23

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090 –1123, aug 1996. 14, 17, 23

[MA00] Bruno Marre and Agnes Arnould. Test sequences generation from lustre descriptions: Gatel. In *Proceedings of the 15th IEEE international conference on Automated software engineering*, ASE '00, pages 229–, Washington, DC, USA, 2000. IEEE Computer Society. 15

[Mac00] Patrícia D. L. Machado. Testing from structured algebraic specifications. In Teodor Rus, editor, *AMAST*, volume 1816 of *Lecture Notes in Computer Science*, pages 529–544. Springer, 2000. 21

[MD98] Brendan Mahony and Jin Song Dong. Blending object-z and timed csp: an introduction to tcoz. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 95–104, Washington, DC, USA, 1998. IEEE Computer Society. 27

[Mer95] Stephan Merz. Mechanizing TLA in Isabelle. In Robert Rodošek, editor, *Workshop on Verification in New Orientations*, pages 54–74, Maribor, July 1995. Univ. of Maribor. 63

[Moo56] Edward F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*, pages 129–153. Princeton U., 1956. 23

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002. 50

[NSM08] Sidney Nogueira, Augusto Sampaio, and Alexandre Mota. Guided test generation from csp models. In *Proceedings of the 5th international colloquium on Theoretical Aspects of Computing*, pages 258–273, Berlin, Heidelberg, 2008. Springer-Verlag. 22

[OCW06] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in proofpower-z. In Steve Dunne and Bill Stoddart, editors, *Unifying Theories of Programming*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer Berlin / Heidelberg, 2006. 62, 63

[OCW07] Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. A denotational semantics for Circus. *Electron. Notes Theor. Comput. Sci.*, 187:107–123, 2007. 36, 43, 62, 75, 86

[Oli06] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science - University of York, UK, 2006. YCST-2006-02. 37, 47

[ON99] David von Oheimb and Tobias Nipkow. Machine-checking the java specification: Proving type-safety. In *Formal Syntax and Semantics of Java*, pages 119–156, London, UK, UK, 1999. Springer-Verlag. 63

[Pau89] L. C. Paulson. The foundation of a generic theorem prover. *J. Autom. Reason.*, 5(3):363–397, September 1989. 51

[PBB98] Cécile Péraire, Stéphane Barbey, and Didier Buchs. Test selection for object-oriented software based on formal specifications. In *Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, PROCOMET '98, pages 385–403, London, UK, UK, 1998. Chapman & Hall, Ltd. 15

[Pel96] Jan Peleska. Test automation for safety-critical systems: Industrial application and future developments. In Marie-Claude Gaudel and James Woodcock, editors, *FME'96: Industrial Benefit and Advances in Formal Methods*, volume 1051 of *Lecture Notes in Computer Science*, pages 39–59. Springer Berlin / Heidelberg, 1996. 22

[Pet01] Alexandre Petrenko. Modeling and verification of parallel processes. chapter Fault model-driven test derivation from finite state models: annotated bibliography, pages 196–205. Springer-Verlag New York, Inc., New York, NY, USA, 2001. 17

[Pha94] Marc Phalippou. *Relations d'implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Universite de Bordeaux I, 1994. 15

[PP] Proofpower web-site. http://www.lemma-one.com/ProofPower/index/index.html. 50, 62

[PVS] Pvs web-site. http://pvs.csl.sri.com/. 50

[RHB97] A. W. Roscoe, C. A. R. Hoare, and Richard Bird. *The Theory and Practice of Concurrency*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. 22, 36, 39

[Rog06] Markus Roggenbach. Csp-casl: a new integration of process algebra and algebraic specification. *Theor. Comput. Sci.*, 354(1):42–71, March 2006. 27

[SAA02] Gwen Salaün, Michel Allemand, and Christian Attiogbé. Specification of an access control system with a formalism combining ccs and casl. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium*, IPDPS '02, pages 303–, Washington, DC, USA, 2002. IEEE Computer Society. 27

[SC96] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Trans. Softw. Eng.*, 22(11):777–793, November 1996. 20

[Sch99a] Steve Schneider. Abstraction and testing. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, pages 738–757, London, UK, UK, 1999. Springer-Verlag. 22

[Sch99b] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999. 22

[Sch01] Steve Schneider. *The B-Method: An Introduction*. Cornerstones of Computing. Palgrave Macmillan, oct 2001. 20

[SCS97] Harbhajan Singh, Mirko Conrad, and Sadegh Sadeghipour. Test case design based on z and the classification-tree method. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM '97, pages 81–, Washington, DC, USA, 1997. IEEE Computer Society. 20

[Sie97] J. Peleska M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 1997. 22

[Sig00] M. Sighireanu. *LOTOS NT User's Manual (Version 2.1)*. INRIA projet VASY, November 2000. 26

[Spi88] J. M. Spivey. *Understanding Z: a specification language and its formal semantics*. Cambridge University Press, New York, NY, USA, 1988. 18

[Spi92] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992. 18, 37

[ST05] Steve Schneider and Helen Treharne. Csp theorems for communicating b machines. *Formal Asp. Comput.*, 17(4):390–422, 2005. 27

[SW09] Norbert Schirmer and Makarius Wenzel. State spaces — the locale way. *Electron. Notes Theor. Comput. Sci.*, 254:161–179, October 2009. 64

[SWC02] Augusto Sampaio, Jim Woodcock, and Ana Cavalcanti. Refinement in circus. In *Proceedings of the International Symposium of Formal Methods Europe on Formal Methods - Getting IT Right*, FME '02, pages 451–470, London, UK, UK, 2002. Springer-Verlag. 45, 90

[SWS] Symbolic web service testing. http://swst.lri.fr/. 170

[TB03] G J Tretmans and H Brinksma. *TorX: Automated Model-Based Testing*, pages 31–43. 2003. 25

[Tre96] Jan Tretmans. Conformance testing with labelled transition systems: implementation relations and test generation. *Comput. Netw. ISDN Syst.*, 29(1):49–79, December 1996. 24

[TS91] Piyu Tripathy and Behcet Sarikaya. Test generation from lotos specifications. *IEEE Trans. Comput.*, 40(4):543–552, April 1991. 26

[TW97] Haykal Tej and Burkhart Wolff. A corrected failure divergence model for csp in isabelle/hol. In *Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, FME '97, pages 318–337, London, UK, UK, 1997. Springer-Verlag. 56, 63

[vABM97] Lionel van Aertryck, Marc Benveniste, and Daniel Le Métayer. Casting: A formally based software test generation method. In *Proceedings of the 1st International Conference on Formal Engineering Methods*, ICFEM

'97, pages 101–, Washington, DC, USA, 1997. IEEE Computer Society. 20

[VDM] Vdm portal web-site. http://www.vdmportal.org/. 18

[vdSU95] Hans van der Schoot and Hasan Ural. Dataflow oriented test selection for lotos. *Comput. Netw. ISDN Syst.*, 27(7):1111–1136, May 1995. 26

[VPK04] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004. 31

[WC02] Jim Woodcock and Ana Cavalcanti. The semantics of circus. In *Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '02, pages 184–203, London, UK, UK, 2002. Springer-Verlag. 36, 43

[WCGF07] Jim Woodcock, Ana Cavalcanti, Marie-Claude Gaudel, and Leo Freitas. Operational Semantics for Circus. *Formal Aspects of Computing*, 2007. 43, 100, 103, 121, 169

[WD96] Jim Woodcock and Jim Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 18

[ZC09a] Frank Zeyda and Ana Cavalcanti. Encoding Circus programs in ProofPowerZ. In *Unifying Theories of Programming, Second International Symposium, UTP 2008, Trinity College, Dublin, Ireland, September 8-10, 2008, Revised Selected Papers*, volume 5713 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009. http://www.cs.york.ac.uk/circus/publications/docs/zc09b.pdf. 75

[ZC09b] Frank Zeyda and Ana Cavalcanti. Mechanical reasoning about families of utp theories. *Electron. Notes Theor. Comput. Sci.*, 240:239–257, July 2009. 62

[ZC10] Frank Zeyda and Ana Cavalcanti. Encoding circus programs in proofpower-z. In *Proceedings of the 2nd international conference on Unifying theories of programming*, UTP'08, pages 218–237, Berlin, Heidelberg, 2010. Springer-Verlag. 62