Graphes de localités : une approche formelle à l'encapsulation et implémentation

Martin Vassor

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Martin VASSOR

Thèse dirigée par **Jean-bernard STEFANI**

préparée au sein du **Laboratoire Institut National de Recherche en Informatique et en Automatique**
dans **l'École Doctorale Mathématiques, Sciences et technologies de l'information, Informatique**

Graphes de localités : une approche formelle à l'encapsulation et implémentation

Location graphs: a formal approach to encapsulation and its implementation

Thèse soutenue publiquement le **7 mai 2021**,
devant le jury composé de :

Monsieur JEAN-BERNARD STEFANI
DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES, Directeur de thèse
Monsieur ALAN SCHMITT
DIRECTEUR DE RECHERCHE, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE, Rapporteur
Monsieur GUIANLUIGI ZAVATTARO
PROFESSEUR, Alma Mater Studiorum-Univ di Bologna, Rapporteur
Monsieur FARHAD ARBAB
PROFESSEUR, Universiteit Leiden, Examinateur
Madame FLORENCE MARANINCHI
PROFESSEUR DES UNIVERSITES, UNIVERSITE GRENOBLE ALPES, Présidente

Location graphs

A formal approach to encapsulation and its implementation

Martin Vassor

Acknowledgments

This thesis is the result of a joint work with Jean-Bernard Stefani and Pascal Fradet, which started as an internship and ended up in a Ph.D. thesis.
The manuscript was reviewed by Alan Schmitt and Gianluigi Zavattaro. In addition, the jury was composed of Farhad Arbab and Florence Maraninchi, as president of the jury.
The page setting of this document was inspired from the thesis written by Robin Gibaud [27].

Abstract

Component based systems ease programming, thanks to the ability to compose multiples small and independent atoms into bigger aggregates, reducing the individual complexity of each atom. The counterpart of this programming paradigm is the emergence of new kinds of errors related to the very composition of those elements. Multiple approaches have been proposed in order to certify the correctness of the composition with respect to a chosen policy.

The location graphs framework make the choice to authorise or forbid each component transition, according the topology of the component graph at the time of the transition. This model offers a wide range of policies that can be implemented.

This thesis is formed of two parts: first, we study the notion of encapsulation, inherited from object oriented programming, in the context of component based programming, taking the location graph framework as a substrate; second, we propose an implementation, given as a Rust library, of the location graph framework.

The study of the notion of encapsulation leads us to three main contributions: (i) a new notion of strong bisimulation for location graphs, allowing the comparison of heterogeneous location graph instances; (ii) the exhibition of a strong notion of encapsulation for that model, characterised with a behavioural equivalence; and (iii) the instantiation, for illustration purposes, of multiple encapsulation policies, highlighting both the precision and the diversity of policies available in the location graph framework. Aside the contribution of the implementation itself, we implemented multiple non-trivial examples showing, in practice, how the original framework can be used.

Résumé

Les systèmes informatiques fondés sur des composants facilitent la programmation grâce à la possibilité de composer différents petits atomes indépendants en aggrégats plus gros, réduisant la complexité individuelle de chaque atome. La contrepartie à ce modèle de programmation est l'apparition de nouvelles erreurs liés à la composition de ces élements. Plusieurs approches ont été proposées afin de garantir la correction de la composition par rapport à une politique choisie.

L'approche du modèle des graphes de localités consiste à autoriser ou d'interdire chaque transition de composant, en fonction de la topologie du graphe de composants au moment de la transition. Ce modèle s'illustre par la diversité des politiques qui peuvent être implémentées.

Cette thèse est composée de deux parties : dans un premier temps, nous étudions la notion de d'encapsulation, héritée de la programmation orientée objet, dans le contexte de la programmation par composant, en prenant pour support le modèle des graphes de localités; dans un second temps, nous proposons une implémentation, sous la forme d'une bibliothèque logicielle pour le langage Rust, du modèle de graphes de localités.

L'étude de la notion d'encapsulation nous a conduit à trois contributions principaux : (i) une nouvelle notion de bisimulation forte pour les graphes de localités, permettant la mise en relation d'instances de graphes de localités hétérogènes; (ii) la mise en évidence d'une notion forte d'encapsulation dans ce modèle, caractérisée par une equivalence comportementale; et (iii) l'instanciation, à titre d'exemple, de plusieurs politiques d'encapsulation, illustrant à la fois la finesse et la diversité des politiques disponibles avec les graphes de localités. Au-delà de la contribution que constitue en elle-même l'implémentation, nous avons implémenté plusieurs exemples non triviaux, illustrant, en pratique, une utilisation possible du modèle original.

Contents

Conventions **vii**

1 Introduction **1**
1.1 Encapsulation . . . 1
1.2 A bestiary of policies . . . 4
1.3 Context, goals and scope . . . 7

2 State of the Art **9**
2.1 Isolation in process calculi . . . 9
2.2 Language approach . . . 11
2.2.1 Ownership based approaches . . . 11
2.2.2 Access contracts . . . 15
2.2.3 Programming languages . . . 19
2.3 System approach . . . 20
2.3.1 Law-Governed Interactions . . . 20

3 Theoretical aspects of Location Graphs **23**
3.1 The location graphs model and semantics . . . 23
3.1.1 Elements of location graphs . . . 25
3.1.2 Semantics of location graphs . . . 27
3.1.3 Additional operations . . . 37
3.2 Comparing Location Graphs . . . 39
3.2.1 Simulation relations for location graphs . . . 40
3.2.2 Heterogeneous simulations . . . 41
3.2.3 Partial bisimulation . . . 42
3.3 Nested Location Graphs . . . 43
3.3.1 Nesting functions . . . 45
3.3.2 Semantics of the 2^{nd} order graph . . . 51
3.3.3 Partial bisimulation . . . 58
3.3.4 Multiple levels of nesting . . . 66

4 Encapsulation policies in Location Graphs **73**
4.1 Hierarchical policies . . . 73
4.1.1 Actor Model . . . 73
4.1.2 Shared Encapsulation Policy . . . 80
4.1.3 Multi-Level Encapsulation Policy . . . 82
4.2 Logging system . . . 82

5 Rust implementation of the location graph framework **89**
5.1 Programming model . . . 90
5.1.1 Design choices . . . 90
5.1.2 Divergences from the theoretical framework . . . 92
5.2 An abstract machine for location graphs . . . 94
5.2.1 Preliminary definitions . . . 94
5.2.2 Transitions, locations and local semantics . . . 94
5.2.3 Graphs and global semantics . . . 101
5.3 Rust API . . . 109
5.3.1 Preliminary steps . . . 110

5.3.2 Locations . . . 110
5.3.3 Authorisation functions and unconstrained location transitions . . . 111
5.3.4 Final steps . . . 113
5.4 Implementation . . . 115
5.4.1 Locations and Transitions . . . 115
5.4.2 Skeleton graphs . . . 118
5.4.3 Roles management and message exchanges . . . 119
5.4.4 Transition selection and resolution . . . 121
5.4.5 Authorisation functions and unconstrained location transitions set . . . 127
5.5 Utilities . . . 127
5.5.1 Trivial authorisation function and transition set . . . 130
5.5.2 Generic role names . . . 133
5.5.3 TCP connections . . . 133
5.6 Encapsulation policies in Rust using Location Graphs . . . 137
5.6.1 An application of the owners-as-ombudsmen: a bank system . . . 137
5.6.2 An application of the logger system: a Publish-Subscribe server . . . 141

6 Conclusion **155**
6.1 General conclusion . . . 155
6.2 Future work . . . 156
6.2.1 Short term . . . 156
6.2.2 Perspectives . . . 157

A Rust in a nutshell **159**
A.1 Quick introduction to Rust . . . 159
A.2 Concurrency . . . 161

B Internal Simulation **163**

Conventions

Typesetting conventions

Theorem 1 (Theorems (and anything that requires a proof) are in red)**.** *In this document, all theorems, lemmas, corollaries, etc. are stated in a red environment.*

Definition 1 (Definitions)**.** The main definitions are stated in a blue environment.

Example 1 (Examples)**.** Examples are shown in an green environment.

Conjecture 1 (Conjectures)**.** *Assumptions and conjectures are displayed in a yellow environment.*

Induction Hypothesis (Induction hypothesis). When proofs are difficult to follow (typically when multiple inductions are performed), induction hypotheses are explicitly written. The induction hypotheses introduced thusly ends with a right-aligned, right-pointing triangle. ▷

Remark (Remarks). Remarks are preceded by the keyword "Remark". Remarks end with a small right-aligned, left-pointing triangle. ◁

Remark (Rust). Remarks concerning particularities of the Rust language are preceded with the Rust gearing. ◁

Remarks that are not necessary to understand the remaining of the work, or which goes further (e.g. to explain the rationale of a choice made) are printed in unnumbered sideboxes.

In order to enjoy cross-references and hyperlinks, this document is best viewed in pdf.

This document was typeset using LaTeX with standard packages available from CTAN. In addition, the third party lstlisting-rust package[1] was used to typeset Rust listings.

Grammatical conventions

This document is intentionally written in a gender neutral way. We use *they* to refer to a person of unspecified gender[2].

Example 2 (Gender neutral sentence)**.** The user does what they want.

Also, even though the plural of "lemma" should be "lemmata", we stick to "lemmas".

Standard notations

Notation (Quantified formulae). In quantified formulae, a central dot is used as a separator between the quantification and the following property:

$\forall a \in \mathbb{A} \cdot P(a)$ reads *for all a in the set $\mathbb{A}$, $P(a)$ holds.*

$\exists a \in \mathbb{A} \cdot P(a)$ reads *there exists an a in the set $\mathbb{A}$ such that $P(a)$ holds.*

Notation (Definition). Definition is denoted by $\triangleq$:

$f \triangleq a$ reads *f is defined as a.*

[1] `https://github.com/denki/listings-rust`

[2] `https://dictionary.cambridge.org/dictionary/english/they`

Notation (Tuples). Tuples are noted with angled brackets: $\langle \dots \rangle$. The projection of a tuple t on its i-th element is noted $\pi_i(t)$.

Notation (Multiset). Multiset are denoted with doubled curly brackets: $\{\!\{a, a\}\!\}$.

Notation (List). Lists are noted with squared brackets, elements of the list are separated with a comma: $[a, b]$ denotes a list of two elements a and b.

Notation (Powerset). We note $\mathcal{P}(\mathbb{S})$ the powerset of a set $\mathbb{S}$, $\mathcal{P}^\star(\mathbb{S})$ the set of all multisets with elements in $\mathbb{S}$ (a.k.a. power multiset), $\mathcal{P}^+(\mathbb{S})$ the set of all lists with elements in $\mathbb{S}$.

Notation (Multiset sum). We note $S_1 +^\star S_2$ the multiset sum of S_1 and S_2, i.e. the multiset such that the cardinality of each element is the sum of the cardinality of that element in S_1 and in S_2.

Notation (List concatenation and element addition). Given l_1 and l_2 two lists in $\mathcal{P}^+(\mathbb{S})$ and s an element of $\mathbb{S}$:

(i) we note $l_1 \mathbin{@} l_2$ the concatenation of l_1 and l_2;
(ii) we note $s :: l_1$ for $[s] \mathbin{@} l_1$.

Notation (Partial function). A partial function f from X to Y is noted $f : X \nrightarrow Y$

Notation (Anonymous function). As usual, we use $\mapsto$ arrows to note anonymous function: $x \mapsto y$ denotes the function which associates x to y.

Notation (Symmetric difference). The symmetric difference (or disjunctive union) of two sets A and B is noted $A \ominus B$. $A \ominus B \triangleq (A \cup B) \setminus (A \cap B)$.

Chapter 1

Introduction

Programming is difficult. While I could not find a paper to cite to emphasize this statement, I think every programmer would agree with me. The difficulty of programming is a bit peculiar, compared to other fields. Think of civil engineering: it is a hard job because it requires working with heavy tools, on unfriendly environment, with external constraints. Think of astronomy: it is hard since it deals with a lot of unknown, about objects one can only observe. Programming is different: none of the above difficulties apply to us. Instead, the essence of the difficulty of programming comes from the complex nature of our task, with the original meaning of "complex": "made up of multiple parts; composite; not simple"[1].

Yet, even as programmers, we often forget the difficulty of programming, thanks to decades of work to create tools to ease this job. For a second, let us forget about those tools and consider this basic statement about programming: when we are programming, we are manipulating thousands, if not millions, of elements of various nature. Even programming such a simple thing (compared to other systems) as the font of this document (computer modern) involve more than 60 parameters[2], just to tune the style of the letters (i.e. not including the skeletons of the letters).

As programmers, to tackle this difficulty the two best tools we have are first *abstraction*, that is the possibility to forget details of both what we are using and what we are building; and, second, *semantics*[3], that is the fact that we forbid ourself to do nonsense.

As such, programming tools are mechanisms, concepts, or anything that helps toward those two directions. Think, for instance, of typing: giving an object a type allows us to forget, most of the time, its actual representation, and type analysis prevent the programmer to mix tomatoes and carrots, except when they intend to make soup.

This thesis is about an other tool: *encapsulation*. This introductory chapter shows that, while the concept of encapsulation is well-known among programmers, it is a notion hard to define properly, and we usually intend to add exceptions to its very nature. In the end, the goal of this thesis is to provide a tool better suited to handle this moving nature of encapsulation, and, by transitivity, *to make programming less difficult.*

Contents

1.1 Encapsulation **1**
1.2 A bestiary of policies **4**
1.3 Context, goals and scope **7**

1.1 Encapsulation

The intuition of encapsulation is to partition a program into multiple components, with the hope that each individual component is easier to program (since smaller), and then less error-prone. With the assumption that correctness is preserved under composition, we conclude that the overall program shall be correct.

Historical context. Such approach is not new. In 1972, Parnas published a short paper on the criteria to be used to split a program into modules [44]. This paper shows two possible decompositions for the same system, and emphasize that, among those two decompositions, we should prefer the one that *hides* information, instead of one that focuses on technical aspects (typically based on a flowchart of the system).

[1] `https://en.wiktionary.org/wiki/complex`, on October 21st, 2020.
[2] `https://en.wikipedia.org/wiki/Computer_Modern`
[3] In its common sense, not as in language theory.

```
System simple_cs = {
  Component client = {
    Port sendRequest;
    Properties { ... }}
  Component server = {
    Port receiveRequest;
    Properties { ...
    maxConcurrentClients : integer = 1;
    multithreaded : boolean = false;
    ... }}
  Connector rpc = {
    Role caller;
    Role callee;
    Properties { synchronous : boolean = true;
    maxRoles : integer = 2;
    protocol : WrightSpec = "..." }}
  Attachments {
    client.send-request to rpc.caller ;
    server.receive-request to rpc.callee }
}
```

Figure 1.1: Example of an Acme system. This excerpt describe a simple client-server system: there is one component for the client and for the server, and a connector to link them. The attachment rule specifies that the ports of the component must be attached to the connector. Extract from Figure 3.7 in [25].

The Actor model (which we will explain in more details below), which is often used today, dates from 1973 [31].

Despite being quite an old subject, some questions are still opened, and a consensus on their resolution has not been reached yet, for instance: Is there a general splitting policy? Should the modules boundaries be completely opaque, or could we allow some information to go through? How to achieve non-hierarchical structures?

We shall note that even if each individual component is correct, new errors can appear due to a bad composition. For instance, problems can arise due to an unexpected topology of the graph of components (a component expects to communicate with an other component while it actually communicates with a third one). Therefore, there is a need to express properties on the graph itself, and not only on the individual components. The literature contains various example of such (software) architectures description languages: the Acme framework [25] allows the description of systems based on an ontology of components, connectors, attachments and properties.

As an example, consider the Acme framework. Acme focuses on providing a language for architecture description. *System*s are a graphs whose nodes are *component*s and edges are *connector*s. In addition, one can describe *properties* of systems and components, which may include some *constraint*s (topological, values, etc.). Systems can be nested: a component can actually be a system. Figure 1.1 shows an example of the description of a system architecture. Notice that the actual behaviour of each component (the program which is run) is not described by Acme, but it uses external languages (e.g. C code, in the example this excerpt is taken).

Encapsulation. Nowadays, software engineers work on complex projects. In order to reduce the inherent difficulty to manage such systems, an approach is to split a given system into multiple modules. In order to reason about such system composed of modules, without having to reason about the implementation details of each individual module, we need to abstract those details away and to characterise the (observable) behaviour of such modules: what is relevant to know for a user of the module, and what is not.

Therefore, with modularity comes the question of encapsulation: modularity is the question of dividing a system into sub systems; and encapsulation is the question of how do these sub systems relate, what is publicly available and what should remain private to a module. We can also approach the question of encapsulation by asking the dual question: some information *must* be hidden (e.g. sensitive data), as a designer, how can I ensure that it will not leak?

When we describe component based system, we often implicitly have a notion of encapsulation: we

assume that the modules we describe are formed of smaller modules, that are enclosed and behave according to an (often) informal specification. For instance, in the introduction of [25]:

> [Figure 3.1 of [25]] depicts a system containing a client and server component connected by a RPC connector. The server itself might be represented by a subarchitecture (not shown).

In that example, we assume that the client and the server indeed communicate only via the RPC connector. This is an assumption that, except for the explicitly mentioned communication channels, the modules are opaque.

While they all refer (at least implicitly) to an intuition of encapsulation, none of the papers mentioned above actually study the relation between component based systems and encapsulation, the fact that a sub module is actually equivalent to a single module. Said otherwise, in the quote above, nothing but intuition justifies that the server can actually be "represented by a subarchitecture".

The first part of this thesis is a formal analysis of the notion of encapsulation for component based systems. We adapt a notion of bisimulation to formally characterise the behavioural relation between a system and its encapsulated counterpart.

Encapsulation policy. Depending on the program to be written, some accesses are considered correct and some are not. For instance, consider a login library: at some point, a password is stored in memory, and this storage should not be directly accessible outside the library, to avoid leaking the passwords. Therefore, not all accesses are suitable. In theory, a careful programmer can make a correct system without help from the language. In practice, for large systems, careful enough programmers do not exist, and there is a need for the language to check that all accesses are intended.

Considering again Acme, one can enforce some isolation policies thanks to constraints. Acme provides an expressive language to express those constraints, this small language allows, in particular, the usage of universal and existential quantifiers and it provides primitives to access roles, ports, connectors, etc. of the system. For instance, to express that all connectors should have two and only two roles, one can write the following property[4]: *Forall conn : connector insystemInstance.Connectors @ size(conn.roles) = 2* . As a consequence, we could express strong isolation properties (such as the one based on ownership types), at a high level.

> It seems, reading from [25], that Acme focuses more on being a practical tool and less on providing strong formal semantics. In Chapter 5 of this thesis, we chose to implement location graphs as a library. If we had chosen to write language from scratch, our language for expressing isolation policies would probably have been quite close to those of Acme, which illustrate that both are close to each other.
> Notice, however, a limitation of Acme, which is that it is not possible to *share* components. Components can be implemented as subsystems, which enforces a tree-like structure of components.

Naturally, this raises two questions: (i) which accesses are correct and which are not?; and (ii) how to ensure unwanted accesses are prevented?.

There are (at least) two ways to answer the first question: (i) find some classes of accesses that are usually unwanted; and (ii) dodge the question and let the programmer define itself which accesses they want or not.

The result of the choice changes the approach that can be taken. By selecting a class of allowed accesses, one can optimize its implementation, at the cost of the risk that the chosen policy might not suit the user.

We call this partition of authorised and forbidden accesses an *isolation* or *encapsulation policy*[5].

In the case considered in this thesis, the theoretical framework of location graphs takes the second choice. By being very expressive in terms of possible policies, we have a cost of potentially synchronising every transition.

For the second question, there have been proposals for both static analysis (with type systems) or dynamic analysis (with automatic code instrumentation). Of course, there is a trade-off between the two questions: the more precise the policy, the harder it is to have static analysis; and similarly for the programmer's choice: the more choice they have to define a policy, the harder it is to ensure that policy. When starting this thesis, we did not know which policies would be useful[6]. Therefore, we wanted

[4] Excerpt from paragraph 3.3.3 in [25].

[5] Or simply *policy* for short.

[6] As a matter of fact, we still do not know. As shown in the following section, we know which policies are used in practice, but we also know that those policies are heavily constraint by technical aspects. We first need to have a system which

to remain very generic and to allow as many policies as possible, which implied not to focus on static verification. Therefore, in the context of this thesis, the verification is dynamic.

Dynamic aspects of component-based systems. Dynamic reconfiguration of systems matters. The ability to change the architecture of a given system is appealing. Informally, typical use case are the following:

UPDATES: one has to prove that replacing a component with an updated version does not change the overall behaviour of the system; or, on the other way, that such replacement fixes a bug

FAULT TOLERANCE: one has to model a component that fails, for instance by removing the component from the system

SCALING: one has to create new components to scale up the performances of the overall system.

Therefore, as designers of components based systems, we have to satisfy that need. For instance, the CTRL-F framework [5, 4] was developed to provide a language for such dynamic reconfiguration.

Naturally, dynamic reconfiguration is much more challenging than just configuration checking, as one needs to assert that the properties hold at any time. In such context, we analyse the behaviour of components, in order to show, for instance, that two components have the same behaviour, or that one refines an other, etc. . Therefore, an important part of this thesis is to devise theoretical tools (a form of bisimulation) which allow us to perform this behavioural study.

Conclusion. As a conclusion, there is quite a lot of frameworks to build modular systems. A lot of work has been done to have expressive composition operators, dynamic behaviour of the component graph, and to allow sharing components. However, there is no study on the notion of encapsulation with freedom in the choice of the isolation policy.

The following section illustrates that there is no consensus on a single isolation policy which would fit all needs. Such observation motivates the work of this thesis.

1.2 A bestiary of policies

In this section, we introduce a set of examples of policies we are dealing with. Throughout this thesis, we will take these examples to illustrate the behaviour of our work.

Examples are presented informally. The formalisation of these examples is sometimes subtle, as we will see later. Prior to the presentation of those policies, we have a word about whether encapsulation shall be achieved by constraints on aliases or on accesses.

Access or alias control. A preliminary note is that there are two ways to enforce the policies hereafter. To enforce isolation, one can constraint accesses or aliases.

By access control, we mean that we control the messages that are exchanged, while alias control means that we control which channels exist between components. Each method has its advantages, but, during this thesis, we found that access control was easier to work with than alias control.

In the case of control accesses, the main benefit is that components can freely move across encapsulation boundaries. Indeed, even if, after moving, a component is not allowed to communicate with an other, it can keep existing channels. If the same move happens with an alias control approach, one need to either forbid the move, or remove the channel first[7]. The drawback is, of course, that each message shall be checked individually at exchange.

This duality is yet an other example to illustrate that the very notions of encapsulation and isolation are not well understood, or at least not consensual.

The actor model ([31]). The actor model is a model in which the main entities are *actors* which run concurrently and communicate using messages. Being a locus of computation, actors have access to some memory, which is not shared among actors (i.e. message passing is the exclusive way of communication). An example of an actor system is provided in Figure 1.2.

In practice, actors may be on different machines (in which case memory isolation is ensured by construction), but it is not necessary. Typically, a library can provide an actor model to a language[8], in which case one of the job of that library is to ensure memory isolation; or the actor model can be the computational model of the language[9], in which case the compiler must ensure memory isolation.

allows a wide range of policies, so that users can actually choose the one that suits their need; so that we can observe which patterns emerge.

[7] Notice that, in the general case, this notion of revoking channel can also happen for other reasons than object moves.

[8] e.g. Akka for Scala ([1]), Actix for Rust ([48]) or the C++ Actor Framework (CAF: [14]).

[9] e.g. Erlang

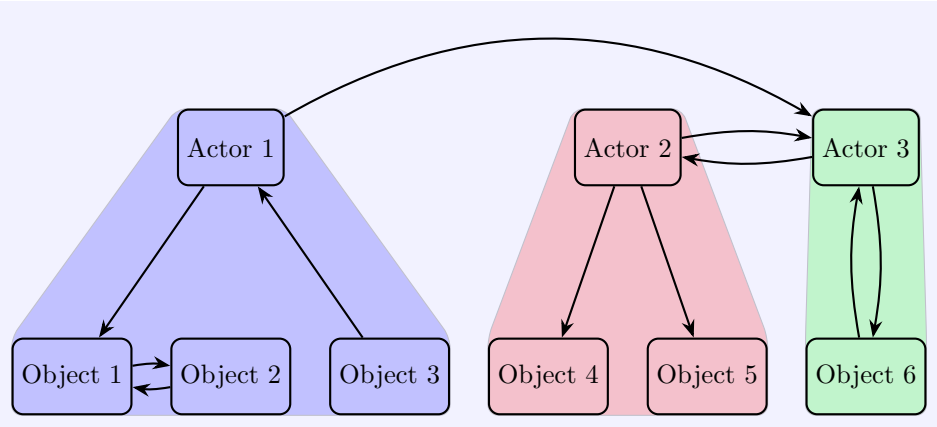


Figure 1.2: Example of an actor system with three actors. Each actor has a private stack of objects used for local computation. Each actor can communicate with objects in its stack or with other actors. Objects can communicate with other objects within the stack they belong to, as well as with their respective actor.

Nested actors. An intuitive extension of the actor model is to nest actors. Said otherwise, objects of an actor are actors themself. Nesting actor comes naturally, for instance to model a cluster of computers running concurrent programs. One may want to have a hierarchy of actors: outer actors to represent distinct computers, and inner actors to represent the concurrent programs. See Figure 1.3 for an example of such system. We consider an arbitrary number of nesting levels.

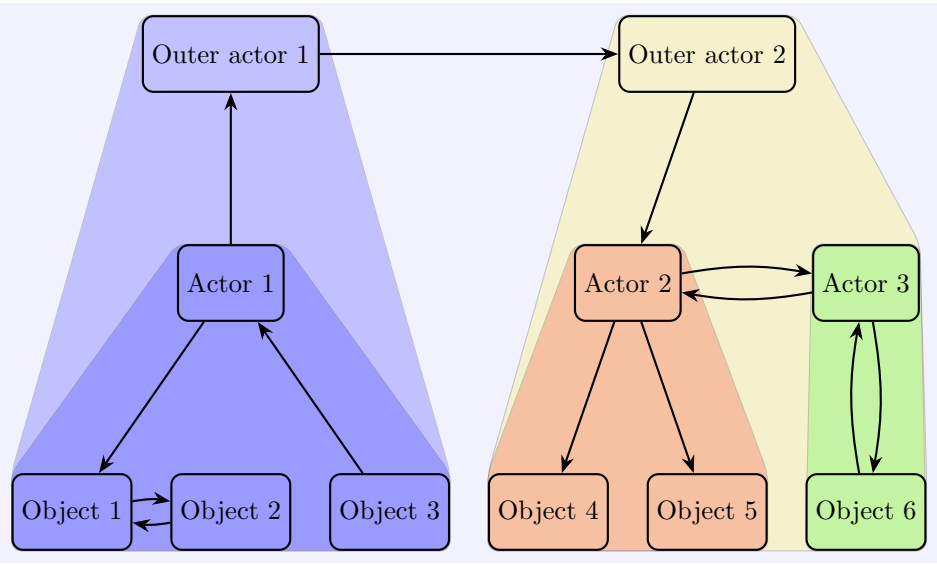


Figure 1.3: Example of nested actors (two levels of nesting). Actors belonging to the same memory domain (e.g. Actor 2 and Actor 3) can communicate as previously, and to their respective higher level actor (Outer actor 2 here). Actors that do not belong (e.g. Actor 1 and Actor 3) to the same domain can not communicate directly (same than objects). Also, notice that an outer actor does not necessarily have a direct link to all inner actors that are in its domain (for instance, in this example, Outer Actor 2 is not directly linked to Actor 3).

A bank management system with shared accounts.

Remark. The two examples discussed originate from [43]. ◁

This example extends the actor model with the possibility to locally break the isolation. Intuitively, we model a bank system in which both Alice and Bob each have a separate account, and in addition they share a third account (see Figure 1.4).

In this example, Alice should be able to communicate with her account and with the shared account, but not with Bob's account (and vice-versa for Bob).

Therefore, this example can not be modeled using the actor model presented before: since Alice should be allowed to modified her account, she must either be an object in the memory of the actor

implementing her account, or an actor at the same level than her account. Since she should also be allowed to modify the shared bank account, she must either be an object in the memory of the actor implementing the shared account, or an actor at the same level of the shared account. Therefore, she can only be implemented as an actor at the same level than both her account and the shared account; and the situation is similar for Bob. All elements are implemented as actors and Alice ends up being able to modify Bob's account[10].

There is a need to have a mechanism to allow relaxing the constraints: to allow objects to belong to multiple actor's memories at the same time.

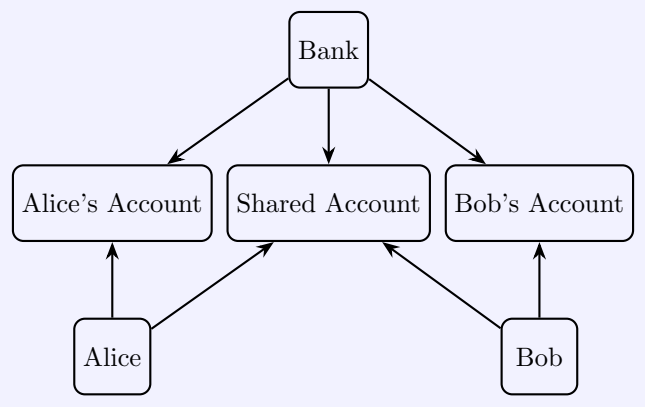


Figure 1.4: An example of a system which requires some notion of sharing.

An other example which illustrates the need for sharing is the example of an iterator for linked lists. Consider a simple linked list, such as the one presented in Figure 1.5. We have a chain of nodes $N_1, \ldots, N_n$, each of them having a reference to their content $D_1, \ldots, D_n$. The user may want to have an iterator to browse the list. In order to be efficient (accesses in constant time), the iterator must have access to the N_is; yet, to be useful, it should not be contained *in* the list aggregate, as it would, in that case, be unavailable from outside the aggregate.

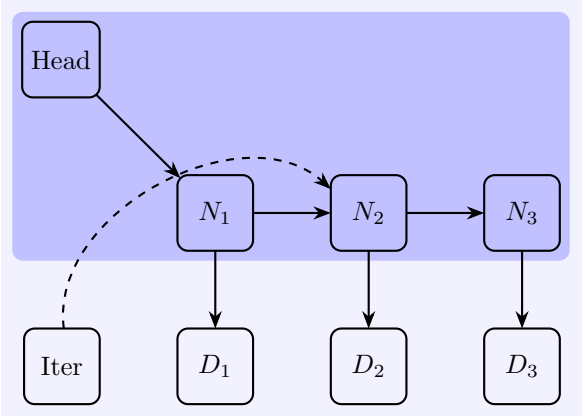


Figure 1.5: A linked list with an iterator. The aggregate of the list is in blue. The iterator object should have an access to the internal details of the aggregates, while remaining outside that aggregate.

A logging system. Consider a software system composed of two parts: the main system S which performs the operations on n subsystems $S_1, \ldots, S_n$; and a logging component L, which offers to access points $L_1, \ldots, L_n$ for $S_1, \ldots, S_n$ respectively (see Figure 1.6 for an example with $n = 2$).

In this system, we want to allow S and L to communicate with their subcomponents, yet we do not want S to be able to communicate with L, nor S_i with L_j and vice-versa, if $i \neq j$.

[10] We do not explore the possibility that Alice and Bob are actors, and the accounts are objects of the actors. This implementation would result in the same situation in which the bank account should lie simultaneously in Alice's and Bob's memories. In addition, with such an implementation, all accounts would not be able to communicate together, which we may, intuitively, want, for instance to implement transfers.

Such systems are useful in practice: logging systems typically write to a single file (or even to the standard or error output). In such cases, L typically implements a queue of logs to be written.

This system does not exhibit any relevant hierarchy. S and L can exist independently, while each L_i exists because of S_i (said otherwise, if S_1 is removed, then L_i should also be removed) and of L (if L is removed, all L_is should be removed too).

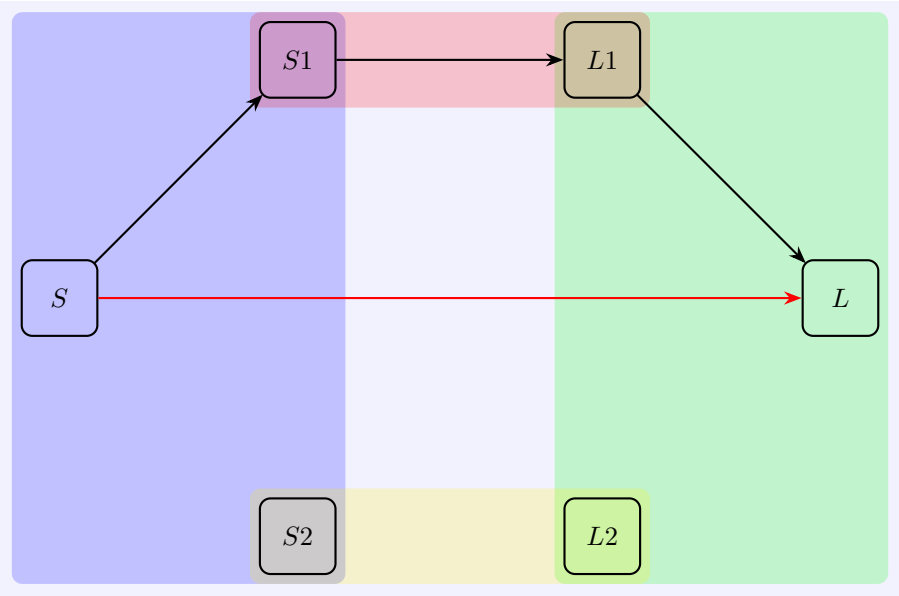


Figure 1.6: An example of the logging system with $n = 2$. Filled areas group nodes that are allowed to communicate together. Channels (i.e. arrows) in black are allowed, since they remain in the same filled area: e.g. the arrow from S to S_1 remains in the blue area. However, channels in red are not allowed, since they do not lie in a single area: e.g. there is not a common area to both S and L, therefore the arrow from S to L is not allowed.

1.3 Context, goals and scope

Context. The starting point of the thesis is the *Location graph* framework[11] (see [57] for an introduction). This framework is explained in further details in Section 3.1; but, in a nutshell, it is a framework for component-based systems which allows the user to define some properties that should be maintained during the execution of an instance. In particular, the policy is not coupled to a given instance, meaning that both part can be written by different groups, and that the policy can be reused across teams, projects, etc.

Goals. As with most component based systems, we want the location graph framework to be suitable to build large software infrastructures. As we saw in the previous sections, to achieve our goal, the notion of encapsulation is of prime interest. In addition, we want it to accommodate various component graph topologies. Therefore, the first goal of this thesis is *to study encapsulation in the location graph framework*, and to show that *the characterisation we will develop can describe all the policies shown in the previous section.*

Concerning this first goal, our contribution is a characterisation of encapsulation with a behavioural equivalence between a plain location graph, and a location graph in which aggregates are implemented as a single location. To the best of our knowledge, this is the first time encapsulation is described with such characterisation. Morally, such approach is interesting as it formally proves that thinking of aggregates of objects as unique composite object is sound. In a second step, we implement the isolation policies described in the previous section, which shows that the framework is flexible enough to accommodate various policies.

The second part of this thesis is devoted to the actual implementation of the location graph framework. Having a actual prototype of the framework is of prime importance, as it allows us to confront our formal framework with actual use cases. It is also a first step to disseminate our (so far) theoretical work to other communities, such as developers or other research teams, and gather feedback from them. Therefore, the second goal of this thesis is *to provide a prototype implementation of the framework.*

[11] Also called *Hypercell*.

Our second contribution is this prototype. Of course, a fully optimised implementation would be the subject of a full thesis, if not more, so we did not intend to optimise the implementation. Yet, it is complete enough to implement the isolation policies we showed. We actually implement two examples which uses those policies.

Outline. Chapter 2 reviews the current techniques and methods that are related to our study. In a first section, we present various formal approaches to the notion of encapsulation in process calculi. In a second section, we review how programming languages currently implement isolation constraints. We conclude this chapter with a small section on a system approach.

Chapter 3 is composed of three parts, in the first one, we introduce the location graph framework, its syntax and semantics. The following section is about the notion of comparison for location graphs. In that second section, we introduce the notion of heterogeneous simulation and partial bisimulation, in order to provide behavioural equivalence among instances of location graphs. Finally, the closing section of that chapter introduces a notion of encapsulation for location graph. This notion is based on nesting graphs, and we show that this notion is consistent with our behavioural equivalence: flat and nested graphs are similar.

Chapter 4 applies the result of the previous chapter to well-known isolation policies. For each policy, we show how to implement it using location graphs, and we formally describe the isolation invariants they ensure. In the first section, we study hierarchical policies, including hierarchical policies with sharing. In the second section, we show an example of non-hierarchical policy: the logging system we presented above.

Chapter 5 presents an implementation of the location graph framework, as a library for Rust. In the first section, we introduce informally the choice we made to implement the framework, in particular the divergences from the theoretical framework. The second section presents an abstract machine, which formally describes our implementation. Section 5.3 presents the library, from a user's perspective. It shows the API provided, and describe informally each component of our library. Section 5.4 is the developer's counterpart to Section 5.3: it presents the internal details of the library and explains how we implemented the abstract machine in Rust. The last two sections of this chapter shows some actual programs we wrote using our library. In Section 5.5, we present some utilities that makes the framework usable in practice. This typically includes an implementation of TCP sockets as locations, etc. Finally, Section 5.6 shows how our running examples can be implemented in our library. This section contains two examples: (i) the example of a simple bank system with clients and (shared) account, to illustrate hierarchical policies with sharing; and (ii) a Publish/Subscribe server, which internally uses a logging system such as the one presented above, to illustrate that our framework can accommodate ad-hoc systems.

For the unfamiliar programmer, a suggested reading order could be, after reading this introduction, to read the introduction of Section 3.1 to get familiar with the ontology of the framework, and then to jump directly to Section 5.3 and play with the library to get familiar with the notions developed in this document. Once familiar, they can have a quick look at remaining of Section 3.1 to have a more formal presentation of the framework and get use to the terminology. They can finish with the remaining parts of Chapter 5. Chapters 3 and 4 are not necessary to understand the implementation and the programming aspects of the thesis.

For the theorist, this thesis can be read in the written order.

Scope. Our goal is mainly to offer the possibility to write as many different policies as possible; more than to optimise the efficiency of the resulting program. Said otherwise, we do not aim to restrict the possibilities of the location graph framework in the name of performance.

Chapter 2

State of the Art

The question of isolation is ubiquitous when modelling complex systems. Consider, for instance the context of (formal) process calculi, where processes should not be allowed to communicate freely, object oriented languages, where aliasing has to be controlled, and software systems, for instance with package managers or complex distributed systems (e.g. servers). This chapter presents, in order, the isolation mechanisms that exist for those various contexts.

We review briefly those different contexts. While a lot of approaches have been proposed, we can not find a proposal which has both a strong formal basis for reasoning; a possibility, for the user, to write its own isolation policies; and which isolation properties are stated globally, and not at the component level.

Contents

2.1 Isolation in process calculi 9
2.2 Language approach 11
2.2.1 Ownership based approaches 11
2.2.2 Access contracts 15
2.2.3 Programming languages 19
2.3 System approach 20
2.3.1 Law-Governed Interactions 20

2.1 Isolation in process calculi

The question of isolation and encapsulation in process calculi is both ubiquitous, yet rarely explicit. Consider, for instance, Milner's *Space and motion of communicating agents* [38]: in the introduction, he mentions a list of important concepts to characterise the behaviour of ubiquitous systems, and "encapsulation" is one of them. The very notion of digraph, the model introduced in that book is, in essence, a model of graphs which nodes are digraphs themself, i.e. a form of nesting.

Yet, that occurrence of "encapsulation" in the introduction is the only in the whole book: while the notion of nesting is intuitive and that encapsulation is cited as an important concept, the study of isolation (the other component of encapsulation) is not done[1].

In this section, we will see various proposals of a notion of encapsulation in process algebras. Most often, this notion of encapsulation is quite weak and informal, and is, essentially, a notion of located and moving processes.

We begin our presentation with the π-calculus (which only has a notion of name restriction), and a distributed variant, the $D\pi$-calculus, which is probably the simplest calculus with a notion of location. We continue with *mobile agents*, which formalise the notion of boundary, which can be nested to create hierarchical structures. We continue with KLAIM, which takes an other approach, with a extension of Linda, which is a model for concurrency based on a shared memory. Finally, we present the Kell calculus, which is an attempt to subsume all previous calculi.

Name restriction in the π-calculus. The standard π-calculus ([39, 40]) includes a construct $(\nu x.P)$ for name restriction, which allows to bind a channel (here x) only in some processes (here P). For instance, in the term $P \parallel (\nu x.(Q \parallel R))$, the name x is bound only in processes Q and R, and appears free in P. Such construct already provide some form of isolation. However, this is quite limited in that only

[1] This is, of course, not an attack to Milner's work. To be fair, the book claim to present the general model of bigraphs, and to study its *pure* version, i.e. without additional constraint.

tree structures can be achieved: it is not possible to have three names x, y and z such that x is shared by P and Q; y by Q and R; and z by R and P.

Distributed π-calculus. The distributed π-calculus [29] ([30] for a short intro, [28] for a fully detailed description), called $D\pi$ is a proposal to introduce the notion of location in process algebras, namely to the π-calculus (the exact variant of the π calculus is a polyadic π calculus, but the results can probably be transposed to other variants). The paper comes with a typing system for capabilities, which we do not present here, for the sake of remaining simple.

In a nutshell, the $D\pi$ calculus binds regular π processes to location, to form *nets* such as: $\ell[\![P]\!] \parallel k[\![Q]\!]$ which denotes a net with to processes P and Q, which are respectively located at ℓ and k.

The $D\pi$-calculus adds new primitives to account for the newly added locations: a primitive $\mathrm{go}\ell.P$, which moves the process to ℓ and continues with P, and $\nu_\ell c$, which replace channel creation with located channel creation: the new channel c is created at location ℓ.

An important point with respect to the semantics is that the rule for communication requires both processes to be located at the same location. However, with the primitive to move processes, it is possible to implement an asynchronous communication across locations.

Finally, the usual structural equivalence is adapted to take into account locations: for instance, $\ell[\![P \parallel Q]\!] \equiv \ell[\![P]\!] \parallel \ell[\![Q]\!]$ or $\ell[\![\nu e.P]\!] \equiv \nu_\ell e.\ell[\![P]\!]$ (if $e \neq \ell$).

An informal note on the type system that comes with $D\pi$: the main idea is to attach *capabilities* to location and channels, such as the capability to read or write on a channel, to create channels, etc. With such typesystem, it is possible to prevent some processes to have undesired behaviour.

As a conclusion, $D\pi$ is a simple way to introduce the notion of location to process calculi. It allows the characterisation of simple systems, but not more complex systems, with e.g. nested localities.

Mobile Ambients. *Mobile Ambients* [12] introduces the notion of *Ambients*, which formalise the notion of domain or boundary. In this calculus, processes are enclosed in ambients, e.g. $n[!P]$, where $n[\dots]$ is the ambient and $!P$ a process (in this case a replication, similar to other process calculi). This calculus adds, in addition, some *capabilities*, i.e. actions to control the ambients. There are three kinds of capabilities: *in* n, *out* n, and *open* n. Informally, the first moves the enclosing ambient into n, the second moves the enclosing ambient outside n, and the last removes the ambient n[2].

For instance, the term $n[in\ m.P]|m[Q]$ contains two ambients, n and m, and the process in n begins with a capability that moves n into m. Such term reduces into $m[n[P]|Q]$. The capability *out* n is the dual: suppose $P = out\ m.P'$, then $m[n[out\ m.P']|Q]$ reduces to $n[P']|m[Q]$. Finally, the capability *open* n removes an ambient n (at the same nesting level). For instance, a term $open\ n.P|n[Q]$ reduces to $P|Q$. Notice that, similarly to the distributed π-calculus, communications are expected to be taken within a common ambient, but the calculus allows to move communicating agents across ambients to have a form of asynchronous communication.

This allows the description of hierarchical structures. Notice that, for all three capabilities, the ambient on which the action is applied has no control. For instance, in the *open* example in the previous paragraph, the ambient n has no control over the capability. A variant, call Mobile Safe Ambient [35], introduces such kind of control. There exist a number of other variants, typically to add typesystems [11], to limit the capabilities [10] (which removes the *open* capability, and replace it by primitives for communication across ambients).

As a conclusion, Mobile Ambients include a mechanism to control modules' boundaries. With such mechanisms, one can create or remove domains, and move processes across domains. Yet, it does not allow shared modules: syntax constraint hierarchical structures (even though the hierarchy can change dynamically).

Klaim. An other approach is the one of KLAIM [19] (see [20] for a brief introduction), a calculus inspired by *Linda* [26]. In Linda, the basis is a collection of tuples (the tuple space), each field of each tuple being an expression or a value. The calculus includes primitives to create and add new tuples to the tuple space, and to retrieve some tuples and use them as parameters.

KLAIM adds a notion of locality to Linda's constructs[3]. In KLAIM, processes are attached to a (physical) locality, such as $s ::_\rho P$, to form *nodes*. Such term indicates that the process P takes place at the location s. In addition, processes can manipulate location in an abstract way, via logical localities, i.e. location variables. The ρ in the term is a (partial) mapping from logical to physical localities. A KLAIM *net* is a parallel composition of multiple nodes.

[2]There are some subtleties on the nesting of the n in each case, but we shall keep it simple and ignore those formal aspects in this presentation.

[3]There are other small changes, but not important for this presentation.

Primitives of the calculus are those from Linda, with locality indications. For instance, the term $\mathbf{out}(t)@\ell$ adds the tuple t in the tuple space at location ℓ.

Remark. Notice an important difference between Mobile Agents and KLAIM: the later allows non-hierarchical structures, such as one including forms of sharing. Actually, the authors of [19] remark (Remark 2.1) that it is quite easy to add nesting to KLAIM nets. ◁

In a nutshell, KLAIM is completely different from the previous calculi. First, it is less influenced by the π-calculus. Also, it allows some forms of sharing, with e.g. shared tuple space, although it does not (formally) allow nesting. Finally, the framework does not include a mechanism to enforce a chosen sharing/isolation policy, meaning that the correctness of such sharing policy rely on the skill of the user.

The Kell calculus. The Kell calculus [8, 54] intends to subsume most previous calculi. It is an higher-order process calculus, with localities.

Terms of the calculus are quite similar to those of any other higher-order process calculi, with two differences: (i) the syntax to receive messages is $\xi \triangleright P$, where ξ is a pattern that should match an actual name (the language and the matching rules of patterns are parameters of the language); and (ii) there are *kell messages*, used for passivation, which have the form $a[Q].R$[4].

The addition of those two elements allows a variety of constructions such as forbidding communication across modules, intercepting messages at a boundary, etc. Yet, even tho the Kell calculus unifies previous calculi, it lacks a method to specify the isolation policy that should be enforced.

2.2 Language approach

The second domain where encapsulation takes an important part is, naturally, programming languages, and in particular object-oriented languages. The main approach consists in static verification of some isolation properties, to prevent uncontrolled object aliasing. This static verification is typically ensured by the type system of the language, and lies on *ownership domains*, which is a class of properties which capture the intuitive notion of encapsulation.

An other approach is the instrumentation of a host language with tools to express some policies, which are then ensured (most often dynamically). To illustrate this approach, we will consider Voigt's *access contracts*.

2.2.1 Ownership based approaches

We first present various approaches that are all based on the notion of *ownership*. This notion aims to formalise the notion of *encapsulation* used in object-oriented languages. This approach was first introduced by Clarke, Potter and Noble in [17], and then modified and extended many times. A survey [16] written by Clarke et al. covers in detail the variants we expose in this section.

In object-oriented languages, an object (e.g. a `Car`) can be composed of other objects (e.g. four `Wheels`). The intuition of *encapsulation* is that, in this situation, no external object (e.g. a `Driver`) should be able to access any `Wheel`. Indeed, if the `Driver` wants to speed up, for instance, it should not be able to call the method `increaseRotationSpeed` of the `Wheels`, but they should call the method `speedUp` of the `Car`, which will call the required methods, be it the `increaseRotationSpeed` of the `Wheels`, but also `checkGasLevel`.

Ownership based approaches are characterised by an *ownership relation* among objects, and an access policy described according to this relation. Two kinds of accesses are distinguished: *reads* and *writes*.

We explore extensively a lot of policy variants of this approach. All those variants are interesting, and there is not a single one that supersedes the other. This illustrates the intrinsic difficulty to formally capture our intuition of encapsulation, and the need to let the user choose the actual policy they want.

For each policy, we have a word about the implementation, even though we are more interested in which policies are implemented than in how they are actually implemented.

Owners-as-dominators. The most basic ownership policy is *owners-as-dominators*. In this policy, each object is possibly owned by an other (unique) object, creating a hierarchy of objects[5]. Each owned object has a single owner, and an owner can have multiple owned objects. Objects that belong to the same owner are called *siblings*. We call the *ownership domain* of an owner the set of all objects it owns, in addition to itself.

On access policy, owned objects are not visible outside their ownership domain, i.e. they are not accessible in any way (see Figure 2.1).

[4] Passivation removes a Kell (i.e. a location) and uses the process it contained in an other process: $a[Q].R|a(x)\triangleright P$ reduces to $R|P\{Q/x\}$.

[5] Objects that are not owned being at the top of the hierarchy.

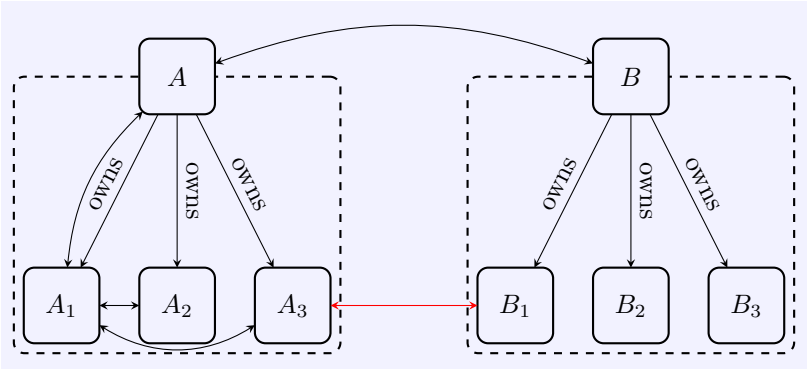


Figure 2.1: Representation of an owner-as-dominator policy: owners (A and B) own A_1, A_2 and A_3 (resp. B_1, B_2 and B_3). Both ownership domains are dashed. Communication between ownership domains are restricted to owner nodes. Communications between owned nodes belonging to different ownership domains (such as A_3 and B_1 in the figure) are forbidden (shown in red in the figure).

This policy was proposed in Clarke et al. seminal paper ([17]) on ownership types. In their paper, the owners-as-dominators policy is statically ensured by extending the Pizza's type system and adding types annotations. Each type is annotated with `rep` or `norep` to indicates whether it belongs to the objects representation or not. An example (taken from [17]) is given in Figure 2.2.

> The actor model [31], in which actors, with isolated internal states, execute concurrently and communicate with asynchronous message-passing primitives is an example of such isolation policy. Sabah's thesis [50] formally proves this fact, by showing that no information can leak from an actor, except by a message sent by the actor.
> Actually, there have been proposals to extend the actor model with shared states. An example is [18]. In this paper, the authors add *domains*, which are special entities which offer two primitives (`whenShared` and `whenExclusive`) to access the state, in a shared or exclusive way. Even though not explicitly said, the authors implement a *owner-as-ombudsmen* (see below).

Owners-as-modifiers. Ownership-as-modifiers is a relaxed version of ownership-as-dominators in which ownership constraints apply only to *write* (i.e. modifying) accesses; *read* accesses are free.

> This policy is interesting as it distinguishes the reference and the usage: objects are allowed to have any references, only using the references is constrained. Despite being quite close to owners-as-dominators, the approach is totally different.

Figure 2.3 shows an example of possible accesses in an owners-as-modifiers context.

Owners-as-ombudsmen ([43]). By design, neither owners-as-dominators nor owners-as-modifiers can be used when some data must be shared, e.g. in the bank example (see Figure 2.4). The owners-as-ombudsmen policy relaxes this constraint by allowing the programmer to explicitly allow some objects to belong to multiple ownership-domains simultaneously.

The owners-as-ombudsmen policy was implemented in a modified Joëlle compiler, a Java based language. In this language, classes are annotated with ownership domains (e.g. `class MyClass<owner, a , b>` has a reference to the ownership domain of its owner, and two other ownership domains `a` and `b`). To share some ownership domains, objects can use the special domain `bridge` to refer to an other object belonging to the same aggregate, and the special domain `aggregate`, which is the domain accessible by all objects refered by `bridge`.

Notice that it does not seem possible for an object to belong to two different aggregates simultaneously.

Figure 2.5 shows an example of the class `Person` used to implement the shared bank account example using Joëlle.

Flexible dynamic ownership. An other approach to relax ownership is *Flexible dynamic ownership* [36, 62]. This is a variant of a dynamically verified owners-as-dominators policy, where the authors add *Filters* and *crossing handlers*.

Intuitively, for each ownership boundary, we can define *in* or *out* filters, which define whose messages can be sent to (resp. from) objects in the ownership domain.

```
1  class Car {
2    rep Engine engine;
3    norep Driver driver;
4
5    // ...
6
7    rep Engine getEngine() { return engine; }
8    void setEngine(rep Engine e) { engine = e; }
9  }
10
11 class Main {
12   void main() {
13     norep Car car = new norep Car();
14     rep Engine e = new rep Engine();
15     norep Driver driver = new norep Driver();
16
17     car.driver = driver; // Allowed
18     car.getEngine().stop(); // Forbidden
19     car.setEngine(e); // Forbidden
20   }
21 }
```

Figure 2.2: Example of the modified Pizza language which implements the owners-as-dominators policy. This example shows the implementation of a `Car` which has two fields: an `Engine` and a `Driver`. The `Engine` is part of the representation of the car (i.e. each instance of `Car` owns its respective `Engine`) while the `Driver` is outside the representation of the `Car`. The `Engine` class (not shown) offers a method `void stop()`. The `main` function instanciates a `Car`, and successively tries to access its fields. Line 17 is allowed, since the `Driver` is not part of the ownership domain of the `Car`, and is therefore accessible. On the other hand, both lines 18 and 19 fail: the former because the `getEngine` method leaks the representation reference (notice that accessing directly the engine with `car.engine.stop()` would also fail); and the latter because the `rep Engine e` belongs to the `Main` representation (`rep` in `Main` and in `Car` do not bind the same ownership domain). Extracted from Figure 1, in [17].

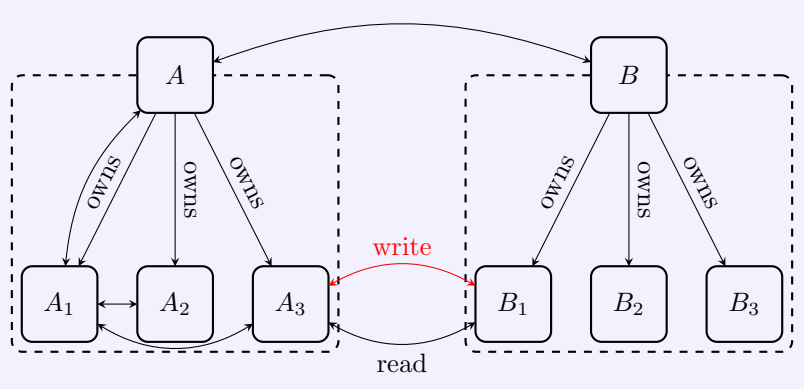


Figure 2.3: Representation of an owner-as-modifiers policy: compared to Figure 2.1, objects can cross ownership domains boundaries to perform read accesses; write accesses remain forbidden. Also, accesses between A and B are possible (both reads and writes) since they do not cross boundary.

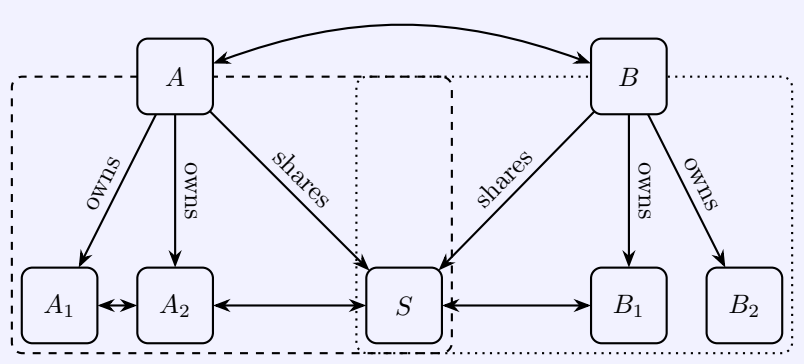


Figure 2.4: Example of a memory layout that can be achieved with the owners-as-ombudsmen policy. The shared object S belongs to the ownership domains of both A (dashed) and B (dotted).

```
class Person<owner> {
  Account<aggregate> account;
  Person<bridge> spouse;
  void share() {
    accout = spouse.getAccount();
  }
  Account<aggregate> getAccount() {
    return account;
  }
}
```

Figure 2.5: Example of an implementation of the bank example using Joëlle implementation of the owners-as-ombudsmen policy. In this example, the `Account` of a `Person` does not lie in the `Person`'s representation, but in a special domain indicated by `aggregate`. This domain is shared among all objects which owner is `bridge` (e.g. the field `spouse`). Extracted from Figure 4, in [43].

On the other hand, crossing handlers dynamically check (when a new reference that crosses an ownership domain is created) that object aliases are controlled. By default, no aliasing is allowed, but the user can customise this behaviour.

Ownership domains. In [2], Aldritch et al. improve ownership policies by allowing the programmer to create and name various ownership domains (even multiple domains per object) and defining access constraints between domains.

This extension separate the definition of ownership domains, which were previously inferred from the structure of the layout of the memory, and access rights from the mechanism which enforce domains isolation. Notice that ownership domains are types annotations, and are therefore checked statically by the typer.

Figure 2.6 shows an example of a linked list implemented using ownership domains. Figure 2.7 shows which objects belong to which ownership domains in that example.

Variations of ownership policies. Ownership-based policies state which accesses to the representation of a given aggregate are allowed. However, there is no constraint on how aggregates can be referenced, nor on whether there can be outgoing arrows from inside an aggregate to the outer world.

External Uniqueness. Some papers suggested coping with aliasing using the notion of unique references[6] (e.g. the *unique* type modifier in [32], unsharable objects in [41]). As shown above, these approaches are to restricted for what we intend to do. However, in [15], Clarke and Wrigstad suggested *External Uniqueness* as a good candidate for the manipulation of aggregates. This policy states that there can be a unique reference to an aggregate from the outer world. Combined with ownership based policies (e.g. owner-as-dominator), this reference can only be toward the owner of the aggregate.

The authors claim that External Uniqueness allows simpler object borrowing and software updates.

Definition of aggregate and outgoing accesses. Ownership policies provide statements on which external objects are allowed to access the internal representation of aggregate objects; they state nothing about accesses from within the aggregate toward the outer world. A simple way to elude the question is to ensure that there can not be outgoing accesses.

For instance, in Island, aggregates are formed as the transitive closure of objects accessible from the aggregate entry point (called the *bridge*).

"An island is the transitive closure of a set of objects accessible from a *bridge* object. A bridge is thus the sole access point to a set of instances that make up an island; no object that is not a member of the island holds any reference to an object within the island apart from the bridge." ([32])

Therefore, by definition, there are no outgoing arrows from the aggregate to the outer world[7].

Balloon types ([3]) follows the same principle, but adds static analysis in order to statically infer and check aggregate isolation.

Notice, in these cases, that the absence of outgoing reference is a direct consequence of the definition of *aggregate*, written as such to simplify (or even perhaps to make possible) the analysis.

2.2.2 Access contracts

The other possibility to introduce isolation control in programming languages is to let the user specify which accesses they allows. We illustrate such choice with *access contracts*.

Presentation. Access contract is a mechanism introduced by Voigt [60, 61] in her Ph.D. thesis. Voigt implemented the mechanism in JavaCon, a modified OpenJDK compiler which rewrite contracts as regular java code.

When a variable is declared, a *contract* is attached to the variable which restricts how the content of the variable can be accessed. Hence, if multiple variables point to the same object (i.e. the object is aliased), multiple contracts exist, each restricting the object access. Notice that aliasing is not prevented, only accesses.

Figure 2.8 shows an example of java with contracts. On line 3, an object `o` is declared with the contracts `true` and `true` (there are two contracts: the first one for read accesses and the last one for write accesses[8]).

[6] There can be at most a unique static reference (class field, heap stored) to each object, yet the reference can be borrowed (i.e. there can exist copies of the reference on the stack, for instance to use it as a method argument).

[7] To be precise, there are no *static* outgoing references (class fields). Dynamic references (i.e. references on the stack) are allowed.

[8] When both contracts are the same, it is possible to write it only once.

```
1  class Sequence<T> assumes owner -> T.owner {
2    domain owned;
3    link owned -> T.owner;
4    owned Cons<T> head;
5    void add(T o) {
6      head = new Cons<T>(o, head)
7    }
8
9    public domain iters;
10   link iters -> T.owner, iters -> owned;
11   iters Iterator<T> getIter() {
12     return new SequenceIterator<T, owned>(head);
13   }
14 }
15
16 class Cons<T> assumes owner -> T.owner {
17   Cons(T obj, owner Cons<T> next) {
18     this.obj = obj;
19     this.next = next;
20   }
21   T obj;
22   owner Cons<T> next;
23 }
24
25 // ...
26
27 class SequenceIterator<T, domain list>
28   implements Iterator<T>
29   assumes list -> T.owner {
30   list Cons<T> current;
31   // ...
32   T next() {
33     T obj = current.obj;
34     current = current.next;
35     return obj;
36   }
37 }
```

Figure 2.6: Example of a linked list with ownership domains. The corresponding memory layout is shown in Figure 2.7. In this example, the class `Sequence` declares two domains: `owned` (line 2) and `iters` (line 9). On line 3, the programmer allows the ownership domain `owned` to access the owner of the type `T`. Similarly, the ownership domain `iter` is allowed to access `T.owner` and `owned` (line 10). One can get an iterator over the sequence by calling the method `getIter`. This iterator belongs to the domain `iter` and can therefore access the elements of the sequence. The `iter` domain being `public`, `Iterators` are accessibles outside their owner's domain. The `Iterator` class, on its side, requires the cells of the list to be allowed to access their content (line 29). Extracted from Figures 3 and 4, in [2].

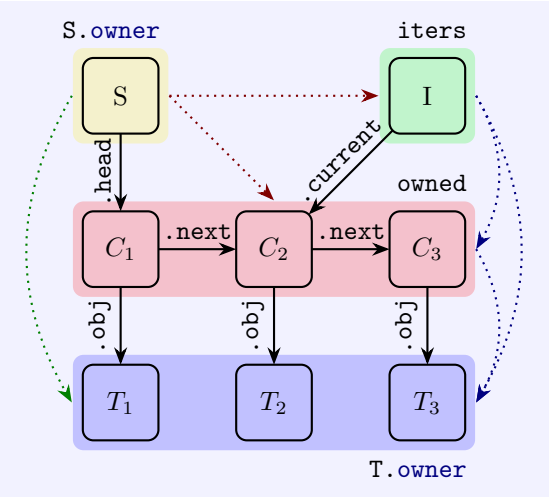


Figure 2.7: Representation of memory layout and ownership domains corresponding to an instance of a linked list presented in 2.6. Initially, we are given the domains `S.owner` (the domain the sequence belongs to) and `T.owner` (the domain the `T`s belong to), and `S.owner` has access to `T.owner`, by assumption (the dotted green arrow represent this assumed clearance). Since `owned` and `iters` are created by S which belongs to `S.owner`, `S.owner` has access to both those domains (the dotted red arrows). Finally, `owned` is explicitly allowed to access `T.owner` and `iters` to access `owned` and `T.owner` (the dotted blue arrows).

```
1 class Main {
2   public static void main(String[] args) {
3     Object o {true, true} = new Object();
4     MyObject o1 {true, true} = new MyObject(o);
5     Object alias {accessor == this ||
6       accessor == accessed} = o;
7     alias.hashCode();
8   }
9 }
```

Figure 2.8: JavaCon: java with contracts. In this example, two objects `o` and `o1` are created, with `o1` referencing `o`. A reference `alias` aliases `o`, and adds new contracts which then forbid `o1` to access `o`.

Contracts are side-effect free expressions which evaluate to boolean. If the expression evaluates to `true`, the access is granted; otherwise it is not. Hence, the contracts of the variable `o` allow any access to the referenced object.

On line 4, `o` is given as an argument to construct `o1`, of type `MyObject`. The reference to `o` is possibly aliased by `o1`. However, the contract described above does not prevent accesses to `o` by `o1`.

When a reference is aliased (such as in line 5), the new contracts are *additional* constraints: all contracts must be satisfied to grant the access. In the example, the contracts of the alias is more restricted than the contracts of `o`: they specify that only `this` or the object can perform the accesses (the `accessor` (resp. `accessed`) keyword is an extension of the language and binds, at the evaluation of the contract, to the object that performs the access (resp. is accessed)). Thus, as soon as the alias is created, `o1` can no longer access `o`, as it would violate the new contracts specified by the `alias` reference.

In addition, there is the possibility to aggregate different references in *groups*, and to use group inclusion to define the contracts. Groups can be manipulated as regular references, and thus allow to manage accesses that are not statically known. The example Voigt present is the case of a linked list, reproduced in Figure 2.9.

In this example, each node of the linked list creates a group `nextNodes` that contains itself (`this`) and all the nodes –transitively– referenced by next. This is possible by recursively including `next.nextNodes` in `this.nextNodes`. Groups can then be used in contracts, using the `in` keyword (to check if a reference belongs to a group), such as in line 3; or the `canread` (resp. `canwrite`) (to check if a reference can read (resp. write) an other reference), such as in line 9.

The contracts mechanism also adds several constructs such as *contracts parameters* (to manipulate

```
1   class LinkedList {
2     group allNodes = {this, head.nextNodes};
3     Node head {accessor == this || accessor in allNodes};
4     // ...
5   }
6
7   class Node {
8     group nextNodes {this, next.nextNodes};
9     Node next {accessor canread this, accessor canwrite this};
10    // ...
11  }
12
```

Figure 2.9: Linked list in JavaCon. All elements which belong to the list are grouped together in the group `allNodes`, which can be used as a regular identifier is contracts. Extracted from [61], p. 49.

```
1 class MyObject {
2   Object ref {false, false};
3   public MyObject(Object ref {true, true}) {
4     this.ref = ref;
5   }
6 }
7
```

Figure 2.10: Implementation of the class `MyObject` used in Figure 2.8. At instanciation, the constructor aliases the object received in argument, with non satisfiable contracts. This prevents *any* reference to the object to be used.

contracts like parameters), *contract suspension* (to temporary disable contract verification), distinction between *pure* and *impure* methods (to distinguish whether methods are read-only, or read/write).

Finally, notice that a static analysis is provided to decide whether some contracts are valid or not. However, not all contracts can be statically verified.

Limitations. Although access contracts provide a very fine grained control over accesses, it is not easily possible to implement system wide invariants. In addition, it is not possible to predict the outcome of a program without a knowledge of the whole program.

Consider again the example in Figure 2.8. At line 7, the `main` method in the `Main` object calls the (pure) method `hashCode()` on the reference `alias`. From the perspective of the `Main` object, such an access should be allowed, as the two known contracts (the one attached to the `o` reference and the `alias` reference) allows it.

However, consider the implementation of the `MyObject` class in Figure 2.10 below. The reference received at instantiation is copied in the variable `this.ref` which both contracts are `false`, forbidding all accesses. Assuming that the object referenced by `o1` is not garbage collected (which would happen in more involved examples), the contracts can not be satisfied and, overall, the program can not make forward progress.

Finally, the behaviour of JavaCon is not well described in presence of concurrency.

"Although not described in detail here, our contract library implementation can handle programs with multiple threads. Its methods and data structures are synchronised in such a way that addition, remove and evaluation of contracts work correctly, even when there are concurrent accesses from different threads." ([61], p. 148.)

However, even if methods are synchronised, it is not clear whether contracts checks are not atomic with fields accesses. For instance, consider a program which contains the method in Figure 2.11 which contains a variable `i`; and, concurrently, `o1` executes `i = 3;` and `o2` executes `i = 4;`.

After the two assignments in `o1` and `o2`, `i` should be `4`: if `o1` perform the assignment first, then `o2`, `i` is trivially `4`; if `o2` modifies `i` first, then `o1` can no longer modify `i`, since the contract is equivalent to `i == 0` when `accessor` is `o1`.

However, contracts verification are not atomic with the access. Hence it can be the case that `i` end up being `3`.

```
void m(Object o1, Object o2) {
  int i = 0 {(accessor == o1 && i == 0) || accessor == o2};
  // o1 tries to assign 3 to i and o2 tries to assign 4 to i concurrently.
}
```

Figure 2.11: Method in JavaCon containing a variable concurrently accessed.

```
def makeCaretaker(target) {
  var enabled := true
  def caretaker {
    match [verb, args] {
      if (enabled)  {
        E.call(target, verb, args)
      } else {
        throw("disabled")
      }
    }
  }
  def gate {
    to enable() { enabled := true }
    to disable() { enabled := false }
  }
  return [caretaker, gate]
}
```

Figure 2.12: Example of the Redell's caretaker pattern in E, from Figure 9.2 in [37]. When the user calls `makeCaretaker`, it instanciates an object with a single field `enabled` and returns a caretaker and a gate. The gate is an object with two methods: `enable` and `disable` which switch `enabled` accordingly. The caretaker object forward each calls to the target, is `enabled` is true.

2.2.3 Programming languages

In this last subsection, we present two programming languages which include a notion of access control in their paradigm. First, we present the E programming language, and then Mezzo.

E. Miller's *E* programming language [37] aims to be secure. Among multiple aspects, we are interested on the *capability* aspects of the language.

The base model of E is the *object-capability* model, which differs from the traditional object model in that it explicitly prohibit some aspects: forged pointers, direct access to (another) object's private stack, mutable static states. Objects communicate together using messages. The idea behind those prohibited primitives is that access to another object can only be granted, not forged.

Notice that the model does not explicitly provide a mechanism to revoke a granted access: if Alice gives Bob a reference to Carol, Alice can not revoke that access after. However, using proxies, one can achieve such revocation: Alice gives Bob a reference to a proxy, that forward messages to Carol. Alice can eventually invalidate the proxy, which stops to forward the messages (called Redell's Caretaker Pattern).

Unfortunately, Miller's thesis is quite vague on the techniques used to enforce those mechanisms. In particular, it does not provide algorithms used to verify, e.g. , $*$-properties.

As of today, the interest of object capabilities increases, in particular for web applications (due to the untrusted nature of such applications). For instance, in [58], Swasey et al. develop a logic for object capability patterns, and formally prove the expected properties, in the context of a concurrent language with closures and mutable states. In [22], Devriese et al. present a similar work, although not formally proved, but on an actual language: Javascript (or at least on λ_{JS}, and already existing core calculus for Javascript).

Mezzo. Mezzo [45] is a programming language, based on the ML language family, with an improved typesytem, compared to other ML languages. The typesystem is based on *permissions*. For instance, the user can declare *mutable* references, which then behave like linear types, or *duplicable* data, which (as the name suggest) means that the data can be copied.

```
data list a =
  | Nil
  | Cons { head: a; tail: list a }

val rec append [a] (consumes (xs: list a, ys: list a)) : list a =
  match xs with
  | Nil -> ys
  | Cons { head = head; tail = tail } ->
    Cons{ head; tail = append (tail, ys) }
end
```

Figure 2.13: Example of Mezzo, adapted from Figure 3.8 of [45]. This example shows the implementation of a list and a function to append two lists. As usual, a list is a sum type, with two variants: an empty list or a concatenation of a head element and a tail list; and the `append` function recursively matches the first list.

The main aspect of Mezzo is that the types are statically checked, even though they are quite expressive. Of course, the trade off is that it is not possible to have guards such as those provided by access contracts.

To give a taste of Mezzo, let show an example, in Figure 2.13, taken from Protzenko's thesis, which illustrate the capabilities of the language. In this example, the `append` *consumes* the two lists. This enforces the type checker to assert that each time the function is called, the two lists passed as parameters can be consumed. To perform this verification, internally, the type checker assign tokens to variables, which keep track of the capabilities of each of the variables. When a function is consumed, the type checker ensures that it is not used anymore. In our example, when `append` is called, the arguments are not available in the caller's scope anymore, and, of course, become available in the callee's scope.

2.3 System approach

In the context of software systems, isolation is also a property that is important. For instance, in the context of security, we are interested in showing the absence of interference between some components, or some topologic invariants.

Law-Governed Interaction is a framework for distributed component systems, which introduces some *guards*, to filter the messages exchanged.

2.3.1 Law-Governed Interactions

Minsky and Ungureanu introduced the *Law-Governed Interactions* framework (*LGI* for short) in [42]. This framework is (originally) made to build asynchronous message-passing distributed infrastructures, whose messages are governed by some user-specified rules.

Presentation. An LGI is a set of agents $A = \{a_1, \ldots, a_n\}$, each exhibiting a control state $C = \{s_1, \ldots, s_n\}$ which exchange messages taken from a set M according to a law L. The tuple $\langle L, A, C, M \rangle$ is called an $\mathcal{L}$-group and is an instance of a LGI.

How a law is written is not important (the authors themself take a Prolog-like language). The important part is that a law is a partial function on event[9] which returns *ruling*, which are actions to be performed on the local control state when the corresponding event is taken.

To implement LGI, each actor a_i is guarded by a *controller*, which is a trusted agent that intercepts messages (and forward them if allowed) and maintain the control state of a_i. Figure 2.14 shows a very simple system with two agents a_1 and a_2, guarded by c_1 and c_2.

The law is a set of rules. Upon event trigger, the associated rule is locally checked, and its effects (if any) are applied to the control state of the actor. An example of law is given in Figure 2.15.

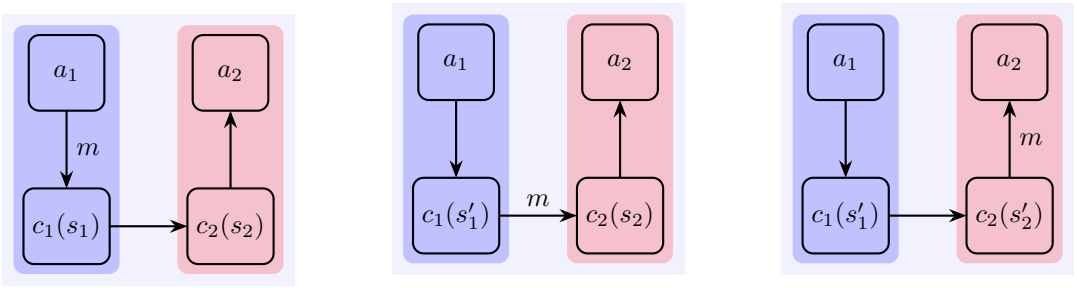
Since rules are locally checked and do not depend on a shared state, the system remains concurrent.

Variants. Over time, variants of the original LGI framework were proposed:

— In [55], Serban and Minsky adapt LGI to synchronous systems;

— In [56], they propose a way to change the law at runtime

[9] The nature of events is not specified, except that messages exchanges (sending and delivering) are events.



(a) A message m sent by a_1 is intercepted by c_1.

(b) The controller c_1 updates the control states and forwards the message to the controller of the receiver.

(c) The controller c_2 updates the control state of the receiver and forwards the message.

Figure 2.14: Example of a Law-Governed system with two actors a_1 and a_2 guarded by two controllers c_1 and c_2, which maintain the control states s_1 and s_2. If a_1 sends a message m to a_2, this message is intercepted by c_1 (Figure 2.14a). If the emission is allowed, the message is forwarded to c_2, and c_1 updates the control state s_1 to s'_1 according to the system law (Figure 2.14b). When c_2 receives the message, it checks whether the reception is allowed, and if it is the case, the message is forwarded to a_2 and the control state s_2 is updated to s'_2 according to the system law (Figure 2.14c).

```
1 R1. arrived(x, m, y) :- amount(A)@CS,
2   do(incr(amount(A))),
3   do(deliver).
4
5 R2. sent(x, m, y) :- do(forward).
```

Figure 2.15: Example of the law of a system. Each control state has a field `amount(n)`, which counts the number of message received. Rule `R1` states that upon reception of a message `m`, this counter shall be increased (at the control place), and the message delivered. Rule `R2` simply states that message emission is free (the message is just forwarded by the controller).

Limitations. Similarly to Access Contracts, rules which compose the law of a system are local. Considering the user wants to enforce a global invariant, they have to find the local rules suitable for the global invariant to hold as a emerging behaviour.

A second limitation is that rules control events (mainly messages, although no exhaustive list of event is presented). We can not, natively, write rules such as: *if x can communicate with y, then x can communicate with z*. Of course, it should be possible, to some extend, to emulate such a behaviour by maintaining an actor graph in the control state shared among all controllers. However, such an emulation would limit the concurrency of the system.

Finally, there is no possibility to bundle multiple events so that all of them succeed or fail. For instance, it is not possible to attempt the sending of two messages at once.

Summary – State of the art comparison

We presented various approaches to isolation and encapsulation. As a conclusion, we show how they relate to each other on various aspects.

Encapsulation. Does the mechanism focuses on the specific problem of encapsulation, or is it a side effect of a generic isolation mechanism? Among the approaches we presented, Ownership types and Acme tackle that problem. The generic approach of Access contracts makes it possible to provide encapsulation, although it is not necessarily the prime goal. On the other hand, LGI is simply a mechanism of access control among agents.

Genericity of the policy. Whether a given mechanism enforces a fixed isolation policy, or whether that policy can be chosen? Variants of Ownership types each enforce a fixed policy, whereas other approach allows the user to chose the policy they want.

Global or local invariants. Whether the isolation invariant is specified for each individual object, or whether it is a global invariant. In Access contract, contracts have to be defined for each object. Similarly, in LGI, rules are local to each agent. On the other hand, in Ownership types and Acme approaches, the invariant is expressed in a global way.

Static or dynamic analysis. Whether the policy is verified statically or dynamically. Ownership types variants are the only one that are verified statically.

Concurrent setting. Whether each mechanism works in a concurrent setting. Ownership types and Acme allow concurrent implementation, although the specification is independent. LGI is a mechanism for concurrent settings. Finally, Access contracts claim to remain correct in concurrent settings, but the thesis does not emphasize this aspects, and the few explanation given makes this claim unclear.

Location graph positioning. With respect to the previous comparison points, the location graph framework takes place as follow. Encapsulation is a problem intended to be addressed. The framework includes specific mechanisms to allow a wide range of encapsulation policies, which can be expressed at the object graph level, not at the boundary of every node, while preserving the capacity to express fine grained policies. The counterpart of this is that the analysis required to enforce the chosen policy is performed dynamically. Finally, being design for component-based systems, the framework was made for concurrent settings.

Chapter 3

Theoretical aspects of Location Graphs

In this chapter, we present the theoretical framework used in this thesis, called the *Location Graph* framework. This framework, first presented as a short paper in [57] and studied more deeply in the draft [33] provides a model to study computations such as the one presented in the previous chapters.

Our goal is to be able to formalise a notion of *encapsulation* using these location graphs. More precisely, we are trying to develop a systematic method to transform an initial graph G into a graph G' whose nodes contain subgraphs of G, while preserving the semantics of G.

This model has a classical notion of *bisimulation*, which allows to compare graphs. Unfortunately, its definition is too restrictive, since we intend to compare graphs that are not directly comparable (we have a graph of processes on one side and a graph of subgraphs on the other side). We hence need to define a more general notion of bisimulation.

In section 3.1, we present the location graph model. In section 3.2, we present our notion of bisimulation. We present two ways to extend regular bisimulations, and we show that they are equivalent. Finally, in section 3.3, we present our nesting method and we show, using our notion of bisimulation, that it preserves the semantics of the original graph.

Contents

3.1 The location graphs model and semantics 23
3.1.1 Elements of location graphs 25
3.1.2 Semantics of location graphs 27
3.1.3 Additional operations 37
3.2 Comparing Location Graphs 39
3.2.1 Simulation relations for location graphs 40
3.2.2 Heterogeneous simulations 41
3.2.3 Partial bisimulation 42
3.3 Nested Location Graphs 43
3.3.1 Nesting functions 45
3.3.2 Semantics of the 2^{nd} order graph 51
3.3.3 Partial bisimulation 58
3.3.4 Multiple levels of nesting 66

3.1 The location graphs model and semantics

Remark. Parts of this section are presented for the context but were done prior to the thesis. ◁

Motivation. The location graph framework is an attempt of to unify various component-based systems, i.e. the framework should be expressive enough so that we can implement various component-based systems.

Our calculus should accommodate mechanisms developed to adapt process calculi to distributed systems. This includes, for instance, localities, a notion introduced to represent the physical machines or logical units which host processes [13], or Cardelli and Gordon's *Mobile Ambients* ([12], subsequent papers [10, 35] are also relevant with respect to this thesis), which introduces a notion of boundary, that processes may or may not cross, according to capabilities (such as "can enter" and "can exit").

We want to provide a formal component model. A lot of component models are quite informal, which causes problems when one has to reason about them. That being said, formal models do exist. For

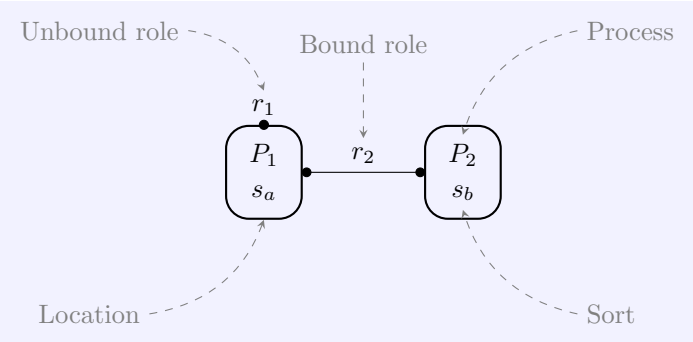


Figure 3.1: Ontology of location graphs. Notice that, graphically, we do not distinguish role directions.

instance, frameworks like *Ptolemy* ([23, 46]) or *BIP* (see [7, 9] for a gentle introduction to BIP). Ptolemy focuses on the heterogeneity of the components (typically, in a single system, a component can have discrete semantics, and an other continuous semantics), through a hierarchical structure. BIP focuses on the notion of *glues*, which are operators to glue components together; various behaviours can be achieved with different glues.

Finally, we are interested in dynamic component models, that is we should include a way to change the topology of the graph at runtime. Various example exists. As one of many instances, let cite *graph rewriting* systems (e.g. [21] or more recently [34]), which describe the relations among components and their evolution in time.

Therefore, in this first section, we present the location graph framework, which covers all the aspects of component-based systems we mentioned above.

Informal presentation. The location graph framework is a model of computation for component based systems. As with most framework for component based systems, the basic ontology contains the two following entities: (i) *components*, which we call *locations* in the context of location graphs, and which are places of computation; and (ii) *roles*, which are endpoints of communication mediums, that locations can bind to communicate together.

Locations are composed of a process, a sort, and of (bound or unbound) roles. A given role can be bound by (at most) two locations at any time. To prevent more than two locations to bind the same role, we use *role directions*, either *required* or *provided*, and we require that each role is bound at most once in a given direction. The direction of the binding have no effect, in particular, it does not constrain the direction of communications. In addition, communication are performed on channels, for a given role. In this thesis, we do not use channels extensively, and therefore, we do not develop further their usage. Figure 3.1 shows the various elements of a simple location graph.

The dynamic aspects of location graphs are governed by three elements: (i) an *unconstrained semantics*, defined according to reduction rules which define how a single location can reduce; (ii) an *authorisation function*, which acts as an oracle which, given the shape of the graph, allows individual location to perform their transition; and (iii) a *global semantics*, which uses the two above elements and defines a rule for composition. Except for the authorisation function, the two other elements are quite usual for component based systems.

As its name suggest, the authorisation function of a location graph instance authorises (or forbids) each individual location transitions. This function rely on the topology of the graph, i.e. the location graph, without the processes, which we call the *skeleton* of the graph. In the skeleton, the roles and the sorts of locations are still visible, and, with the attempted transition, are the only information the authorisation function can use.

As such, each authorisation function defines a *policy*. For the same unconstrained semantics, two authorisation functions can yields different results.

This section introduces formally the elements developped above. Section 3.1.1 defines the static aspects of location graphs. Static properties (such as "there are not two locations that bind the same role in the same direction") are introduced thanks to predicates to ensure *well-formedness*. The dynamic aspects are presented in Section 3.1.2. Prior to the definition of the semantic *per se*, we introduce a few elements, such as the skeleton of a graph, or the labels of the transitions of our semantics. Finally, in Section 3.1.3, we can show various straightforward, but useful, operations and results on graphs.

3.1.1 Elements of location graphs

In this section, we recap the main definitions. The location graph framework uses various elements which are more or less independent. We try to clarify the distinction between a location graph (more or less a set of locations), an instance (the transitions locations can fire and the authorisation function), a policy (characterised by the authorisation function)

We first define what are locations and location graphs; then we recap the semantics of location graphs (*unconstrained transitions* and *authorisation function*).

We are given the following sets:

— $\mathbb{P}$ a set of processes;
— $\mathbb{S}$ a set of sorts;
— $\mathbb{R}$ a set of roles;
— $\mathbb{C}$ a set of channels, which contains a special element $\texttt{rmv}$
— $\mathbb{V}$ a set of values.

Definition 2 (Atom)**.** Roles and channels are called *atoms* and form a set $\mathbb{A} = \mathbb{R} \cup \mathbb{C}$.

Prelocations and locations. We first define locations, which are the nodes of our location graphs. Locations are defined from well-formed prelocations, which are a tuple of (i) a process; (ii) a sort (a dynamic type); (iii) provided roles; and (iv) required roles (we distinguish two *directions* for the roles, to emphasize that they can be bound by at most two locations at any time, see below).

Definition 3 (Prelocation)**.** A prelocation is a element of $\mathbb{L}^{\mathtt{p}} = \mathbb{P} \times \mathbb{S} \times \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathbb{R})$.

The elements are accessible via four functions:

Definition 4 (Elements of a prelocation)**.** Given a prelocation $L = \langle P, \sigma, p, r\rangle$, we define:

$$L.\texttt{proc} \triangleq P \qquad L.\texttt{sort} \triangleq \sigma \qquad L.\texttt{provided} \triangleq p \qquad L.\texttt{required} \triangleq r$$

Not all prelocations are interesting. In the following, we only consider prelocations which do not bind the same role in both *required* and *provided* positions. We call those prelocations *well-formed* and we define the predicate $\texttt{WF}(\cdot)$ which characterise them.

Definition 5 (Well-formed prelocations)**.** $\texttt{WF}(L) \Leftrightarrow L.\texttt{provided} \cap L.\texttt{required} = \emptyset$

Locations are well-formed prelocations.

Definition 6 (Location)**.** A location L is a prelocation such that $\texttt{WF}(L)$. The set of well-formed prelocations is noted $\mathbb{L} \triangleq \{L \in \mathbb{L}^{\mathtt{p}} | \texttt{WF}(L)\}$.

Notation. A location $L = \langle P, \sigma, p, r\rangle$ is written $[P : \sigma \lhd p \bullet r]$.

Pregraphs and graphs.

Definition 7 (Location pregraph)**.** The set $\mathbb{G}^{\mathtt{p}}$ of location pregraphs is the set of terms obtained with the grammar $G ::= \emptyset \mid L \mid G \parallel G$ where $\emptyset$ is the empty pregraph and $L \in \mathbb{L}^{\mathtt{p}}$.

Remark. The fact that elements of location pregraphs are taken from $\mathbb{L}^{\mathtt{p}}$ and not from $\mathbb{L}$ is arbitrary. We chose to enforce the well-formedness of locations at the same place than the well-formedness of overall graphs (see Definition 10 below). ◁

Elements of a location pregraph can be accessed via various functions:

Definition 8 (Elements of a location graph)**.**

$$\emptyset.\texttt{prov} \triangleq \emptyset \qquad L.\texttt{prov} \triangleq L.\texttt{provided} \qquad G_1 \parallel G_2.\texttt{prov} \triangleq G_1.\texttt{prov} \cup G_2.\texttt{prov}$$

$$\emptyset.\texttt{req} \triangleq \emptyset \qquad L.\texttt{req} \triangleq L.\texttt{required} \qquad G_1 \parallel G_2.\texttt{req} \triangleq G_1.\texttt{req} \cup G_2.\texttt{req}$$

$$G.\texttt{roles} \triangleq G.\texttt{prov} \cup G.\texttt{req} \qquad G.\texttt{bound} \triangleq G.\texttt{prov} \cap G.\texttt{req} \qquad G.\texttt{unbound} \triangleq G.\texttt{roles} \setminus G.\texttt{bound}$$

$$G.\texttt{pbound} \triangleq G.\texttt{bound} \cap G.\texttt{prov} \qquad G.\texttt{punbound} \triangleq G.\texttt{unbound} \cap G.\texttt{prov}$$

$$G.\texttt{rbound} \triangleq G.\texttt{bound} \cap G.\texttt{req} \qquad G.\texttt{runbound} \triangleq G.\texttt{unbound} \cap G.\texttt{req}$$

Remark. Distinguish the elements $L.\texttt{provided}$ of a location and $G.\texttt{prov}$ of a location graph. If $G = L$, then $L.\texttt{provided} = G.\texttt{prov}$ (resp. for required roles). ◁

Locations belonging to the same pregraph shall not bind a role twice in the same direction. To ensure that, we define the predicate $\texttt{separate}(\cdot,\cdot)$ which ensure two pregraphs do not have roles in common for each direction.

Definition 9 (Separated location pregraphs)**.**

$$\texttt{separate}(G, G') \triangleq (G.\texttt{prov} \cap G'.\texttt{prov} = \emptyset) \wedge (G.\texttt{req} \cap G'.\texttt{req} = \emptyset)$$

Finally, we only consider pregraphs that are separated and which locations are all well-formed. We call those pregraphs *well-formed* and we define the predicate $\texttt{WF}_G(\cdot)$ to characterise them.

Definition 10 (Well-formed location pregraph)**.**

$$\texttt{WF}_G(\emptyset) \qquad \texttt{WF}_G(L) \Leftrightarrow \texttt{WF}(L) \qquad \texttt{WF}_G(G \parallel G') \Leftrightarrow \texttt{WF}_G(G) \wedge \texttt{WF}_G(G') \wedge \texttt{separate}(G, G')$$

Definition 11 (Location pregraph structural equivalence)**.** Let $\equiv \subseteq \mathbb{L}^{\mathsf{p}} \times \mathbb{L}^{\mathsf{p}}$ be the smallest equivalence relation which includes the following rules:

$$0 \parallel G \equiv G \qquad G_1 \parallel G_2 \equiv G_2 \parallel G_1 \qquad G_1 \parallel (G_2 \parallel G_3) \equiv (G_1 \parallel G_2) \parallel G_3 \qquad \frac{G_1 \equiv G_2}{G_1 \parallel G \equiv G_2 \parallel G}$$

Imported Lemma 1 (Structural equivalence preserves well-formness (Lemma 1 of [33]))*.*

$$\forall G, G' \in \mathbb{G}^{\mathsf{p}} \cdot (\texttt{WF}_G(G) \wedge G \equiv G') \Rightarrow \texttt{WF}_G(G')$$

We can now state that Location Graphs are just well-formed pregraphs, i.e. graphs in which no two locations by the same role in the same direction and in which locations do not bind a role in both required and provided direction.

Definition 12 (Location graph)**.** *Location graphs* are well-formed location pregraphs.
The set of location graphs is noted $\mathbb{G} \triangleq \{G \in \mathbb{G}^{\mathsf{p}} | \texttt{WF}_G(G)\}$.

Remark. Since location graph composition is commutative and associative w.r.t. $\equiv$, we often ignore parenthesis. In addition, we define a notation to note the parallel composition of multiple graphs:

Notation (Parallel composition of location graphs). Given a set of location graphs $\{G_1, \ldots, G_n\}$,

$$\prod_{G_i \in \{G_1,\ldots,G_n\}} G_i = G_1 \parallel \ldots \parallel G_n$$

◁

Remark. We should remember that $\mathbb{L}$ and $\mathbb{G}$ depend on $\mathbb{P} \times \mathbb{S} \times \mathcal{P}(\mathbb{R}) \times \mathcal{P}(\mathbb{R})$. Below, we will work with locations and graphs of different *kinds* at the same time, i.e. we will compare locations and graphs that are not defined using the same sets $\mathbb{P}$, $\mathbb{S}$, $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{V}$. Hence, for such different kinds of locations and graphs, we can have two $\mathbb{L}$ and $\mathbb{G}$ that are not the same. We define the function $\texttt{lgraph}(\mathbb{P}, \mathbb{S}, \mathbb{R})$ which returns the set $\mathbb{G}$ of location graphs defined as above, using $\mathbb{P}$, $\mathbb{S}$, $\mathbb{R}$. ◁

Finally, we define the *support* of a location graph, which simply is the set of all atoms that exist in the graph.

Definition 13 (Support)**.**

$$\mathtt{supp}(\emptyset) \triangleq \emptyset$$

$$\mathtt{supp}([P : s \lhd p \bullet r]) \triangleq \mathtt{supp}_p(P) \cup p \cup r$$

$$\mathtt{supp}(G_1 \parallel G_2) \triangleq \mathtt{supp}(G_1) \cup \mathtt{supp}(G_2)$$

where $\mathtt{supp}_p(P)$ is a function which returns all atoms known by P.

Remark. Notice that $\mathtt{supp}_p(P)$ depends on the actual definition of the set of processes. This is to be defined carefully for each instance; for the rest of this thesis, we suppose it contains all channels and role names contained in the process. ◁

3.1.2 Semantics of location graphs

In this subsection, we give a semantic for location graphs. The semantics relates an *environment* and an (initial) location graph with a *label* and a (final) location graph.

Our notion of environment relies on *skeleton location graphs*, which merely are location graphs with erased processes. First, we define such skeleton location graphs; and after environments.

Transitions of our semantics include *labels*[1] which are, more or less, exchanges of messages and priorities. We define such labels in a third step.

When those elements are properly defined, we can explain the semantics of location graphs. This is done in three steps: (i) we explain the semantics of individual locations; (ii) our graph semantics depends on *authorisation functions*, our second step is to define those; and (iii) we define the location graph semantics. These three steps are described one after each other in the last paragraphs of this section.

Skeleton location graphs. Skeleton location graphs are constructed similarly to location graphs, except that skeleton locations (the counterpart of locations) do not include a process.

Definition 14 (Skeleton location graph)**.** The set of skeleton location graphs is the set of terms obtained with the grammar $G ::= \emptyset \mid [s \lhd p \bullet r] \mid G \parallel G$ where $\emptyset$ is the empty skeleton location graph, and $s \in \mathbb{S}$, $p \subset \mathbb{R}$ and $r \subset \mathbb{R}$.

Remark. The precise reader will, of course, note that the graph $\emptyset$ and the skeleton graph $\emptyset$ are *not* the same. Similarly, the graph composition $\cdot \parallel \cdot$ and the skeleton graph composition $\cdot \parallel \cdot$ are *not* the same constructors.

We made the choice to overload the notations to make reading and getting the intuition easier, at the cost of ambiguity. ◁

Naturally, we are interested in skeletons of given location graphs.

Definition 15 (Skeleton of a location graph)**.**

$$\Sigma(\emptyset) \triangleq \emptyset \qquad \Sigma([P : s \lhd p \bullet r]) \triangleq [s \lhd p \bullet r] \qquad \Sigma(G_1 \parallel G_2) \triangleq \Sigma(G_1) \parallel \Sigma(G_2)$$

The set of skeleton locations is $\mathbb{L}^{\mathtt{s}} \triangleq \{\Sigma(L) | L \in \mathbb{L}\}$ and the set of skeleton graphs is $\mathbb{G}^{\mathtt{s}} \triangleq \{\Sigma(G) | G \in \mathbb{G}\}$.

We define the same accessors for skeleton location graphs than for location graphs.

Definition 16 (Elements of a skeleton location graph)**.**

$$\emptyset.\mathtt{prov} \triangleq \emptyset \qquad [s \lhd p \bullet r].\mathtt{prov} \triangleq p \qquad G_1 \parallel G_2.\mathtt{prov} \triangleq G_1.\mathtt{prov} \cup G_2.\mathtt{prov}$$

$$\emptyset.\mathtt{req} \triangleq \emptyset \qquad [s \lhd p \bullet r].\mathtt{req} \triangleq r \qquad G_1 \parallel G_2.\mathtt{req} \triangleq G_1.\mathtt{req} \cup G_2.\mathtt{req}$$

[1] Notice that, stricto-sensu, our system is not a labelled transition system, since we also have a notion of environment: our transitions are not a subset of $\mathcal{S} \times \mathcal{L} \times \mathcal{S}$ as usual (where $\mathcal{S}$ would be our set of states and $\mathcal{L}$ our set of labels).

$$G.\mathtt{roles} \triangleq G.\mathtt{prov} \cup G.\mathtt{req} \qquad G.\mathtt{bound} \triangleq G.\mathtt{prov} \cap G.\mathtt{req} \qquad G.\mathtt{unbound} \triangleq G.\mathtt{roles} \setminus G.\mathtt{bound}$$

$$G.\mathtt{pbound} \triangleq G.\mathtt{bound} \cap G.\mathtt{prov} \qquad G.\mathtt{punbound} \triangleq G.\mathtt{unbound} \cap G.\mathtt{prov}$$

$$G.\mathtt{rbound} \triangleq G.\mathtt{bound} \cap G.\mathtt{req} \qquad G.\mathtt{runbound} \triangleq G.\mathtt{unbound} \cap G.\mathtt{req}$$

$$[s \lhd p \bullet r].\mathtt{sort} \triangleq r$$

These definitions are, of course, consistent with their graph counterparts.

Lemma 1 (Elements of graph and skeleton graph consistency)**.** *For all location graphs G:*

(i) $G.\mathtt{prov} = \Sigma(G).\mathtt{prov}$
(ii) $G.\mathtt{req} = \Sigma(G).\mathtt{req}$
(iii) $G.\mathtt{roles} = \Sigma(G).\mathtt{roles}$
(iv) $G.\mathtt{bound} = \Sigma(G).\mathtt{bound}$
(v) $G.\mathtt{unbound} = \Sigma(G).\mathtt{unbound}$
(vi) $G.\mathtt{pbound} = \Sigma(G).\mathtt{pbound}$
(vii) $G.\mathtt{punbound} = \Sigma(G).\mathtt{punbound}$
(viii) $G.\mathtt{rbound} = \Sigma(G).\mathtt{rbound}$
(ix) $G.\mathtt{runbound} = \Sigma(G).\mathtt{runbound}$

Proof. Items (i) and (ii) are proven directly, by induction on G:

Case $G = \emptyset$: For provided roles: $\emptyset.\mathtt{prov} = \emptyset$ and $\Sigma(\emptyset).\mathtt{prov} = \emptyset.\mathtt{prov} = \emptyset$. For required roles: $\emptyset.\mathtt{req} = \emptyset$ and $\Sigma(\emptyset).\mathtt{req} = \emptyset.\mathtt{req} = \emptyset$.

Case $G = [P : s \lhd p \bullet r]$: For provided roles: $[P : s \lhd p \bullet r].\mathtt{prov} = p$ and $\Sigma([P : s \lhd p \bullet r]).\mathtt{prov} = [s \lhd p \bullet r].\mathtt{prov} = p$. For required roles: $[P : s \lhd p \bullet r].\mathtt{req} = r$ and $\Sigma([P : s \lhd p \bullet r]).\mathtt{req} = [s \lhd p \bullet r].\mathtt{req} = r$.

Case $G = G_1 \parallel G_2$: Our induction hypothesis is that: $G_1.\mathtt{prov} = \Sigma(G_1).\mathtt{prov}$, $G_2.\mathtt{prov} = \Sigma(G_2).\mathtt{prov}$, $G_1.\mathtt{req} = \Sigma(G_1).\mathtt{req}$, and $G_2.\mathtt{req} = \Sigma(G_2).\mathtt{req}$.
For provided roles: $G_1 \parallel G_2.\mathtt{prov} = G_1.\mathtt{prov} \cup G_2.\mathtt{prov}$ and $\Sigma(G_1 \parallel G_2).\mathtt{prov} = (\Sigma(G_1) \parallel \Sigma(G_2)).\mathtt{prov} = \Sigma(G_1).\mathtt{prov} \cup \Sigma(G_2).\mathtt{prov} = G_1.\mathtt{prov} \cup G_2.\mathtt{prov}$.
For required roles: $G_1 \parallel G_2.\mathtt{req} = G_1.\mathtt{req} \cup G_2.\mathtt{req}$ and $\Sigma(G_1 \parallel G_2).\mathtt{req} = (\Sigma(G_1) \parallel \Sigma(G_2)).\mathtt{req} = \Sigma(G_1).\mathtt{req} \cup \Sigma(G_2).\mathtt{req} = G_1.\mathtt{req} \cup G_2.\mathtt{req}$.

The following items depend on their the previous ones.

Item (iii): $\Sigma(G).\mathtt{roles} = \Sigma(G).\mathtt{prov} \cup \Sigma(G).\mathtt{req} \overset{\text{(i), (ii)}}{=} G.\mathtt{prov} \cup G.\mathtt{req} = G.\mathtt{roles}$.

Item (iv): $\Sigma(G).\mathtt{bound} = \Sigma(G).\mathtt{prov} \cap \Sigma(G).\mathtt{req} \overset{\text{(i), (ii)}}{=} G.\mathtt{prov} \cap G.\mathtt{req} = G.\mathtt{bound}$.

Item (v): $\Sigma(G).\mathtt{unbound} = \Sigma(G).\mathtt{roles} \setminus \Sigma(G).\mathtt{bound} \overset{\text{(iii), (iv)}}{=} G.\mathtt{prov} \setminus G.\mathtt{req} = G.\mathtt{unbound}$.

Item (vi): $\Sigma(G).\mathtt{pbound} = \Sigma(G).\mathtt{bound} \cap \Sigma(G).\mathtt{prov} \overset{\text{(i), (iv)}}{=} G.\mathtt{bound} \cap G.\mathtt{prov} = G.\mathtt{pbound}$.

Item (vii): $\Sigma(G).\mathtt{punbound} = \Sigma(G).\mathtt{unbound} \cap \Sigma(G).\mathtt{prov} \overset{\text{(i), (v)}}{=} G.\mathtt{unbound} \cap G.\mathtt{prov} = G.\mathtt{punbound}$.

Item (viii): $\Sigma(G).\mathtt{rbound} = \Sigma(G).\mathtt{bound} \cap \Sigma(G).\mathtt{req} \overset{\text{(ii), (iv)}}{=} G.\mathtt{bound} \cap G.\mathtt{req} = G.\mathtt{rbound}$.

Item (ix): $\Sigma(G).\mathtt{runbound} = \Sigma(G).\mathtt{unbound} \cap \Sigma(G).\mathtt{req} \overset{\text{(ii), (v)}}{=} G.\mathtt{unbound} \cap G.\mathtt{req} = G.\mathtt{runbound}$. □

We equip the skeleton graphs with a structural equivalence similar to the one of location (pre)graphs.

Definition 17 (Skeleton graph structural equivalence)**.** Let $\equiv \subseteq \mathbb{G}^{\mathsf{s}} \times \mathbb{G}^{\mathsf{s}}$ be the smallest equivalence relation which includes the following rules:

$$0 \parallel G \equiv G \qquad G_1 \parallel G_2 \equiv G_2 \parallel G_1 \qquad G_1 \parallel (G_2 \parallel G_3) \equiv (G_1 \parallel G_2) \parallel G_3 \qquad \frac{G_1 \equiv G_2}{G_1 \parallel G \equiv G_2 \parallel G}$$

Lemma 2 (Composition of skeleton locations)**.**

$$\forall G \in \mathbb{G},\ \prod_{L_i \in G} \Sigma(L_i) \equiv \Sigma(G)$$

Proof. By structural induction on G:
Case $G = \emptyset$: Vacuously holds.

Case $G = [P : s \lhd r \bullet p]$: Trivially holds.

Case $G = G_1 \parallel G_2$:

Induction Hypothesis (IH). $\prod_{L_i \in G_1} \Sigma(L_i) \equiv \Sigma(G_1)$ and $\prod_{L_i \in G_2} \Sigma(L_i) \equiv \Sigma(G_2)$

$$\prod_{L_i \in G} \Sigma(L_i) \equiv \prod_{L_i \in G_1 \parallel G_2} \Sigma(L_i) \equiv \prod_{L_i \in G_1} \Sigma(L_i) \parallel \prod_{L_i \in G_2} \Sigma(L_i) \overset{\text{IH}}{\equiv} \Sigma(G_1) \parallel \Sigma(G_2) \overset{\text{Def 15}}{\equiv} \Sigma(G)$$

□

Lemma 3 (Composition of skeleton locations graphs)**.** *Let $\mathbb{G}_f$ be a set of location graphs such that $G = \prod_{G_i \in \mathbb{G}_f} G_i$ is a location graph.*

$$\prod_{G_i \in \mathbb{G}_f} \Sigma(G_i) \equiv \Sigma(\prod_{G_i \in \mathbb{G}_f} G_i)$$

Proof.

$$\begin{aligned}
\prod_{G_i \in \mathbb{G}_f} \Sigma(G_i) &\overset{\text{Lem 2}}{\equiv} \prod_{G_i \in \mathbb{G}_f} \prod_{L_i \in G_i} \Sigma(L_i) \\
&\overset{\text{Def 17}}{\equiv} \prod_{L_i \in \prod_{G_i \in \mathbb{G}_f} G_i} \Sigma(L_i) \\
&\overset{\text{Hypothesis}}{\equiv} \prod_{L_i \in G} \Sigma(L_i) \\
&\overset{\text{Lem 2}}{\equiv} \Sigma(\prod_{L_i \in G} L_i) \equiv \Sigma(G)
\end{aligned}$$

□

Finally, we define the notion of inclusion and union for skeleton locations graphs.

Definition 18 (Inclusion (Skeleton Location Graph))**.** A skeleton location graph G_1 is included in G_2 (noted $G_1 \subseteq G_2$) if and only if:
(i) If $G_1 \equiv L \parallel G_1'$, then $G_2 \equiv L \parallel G_2'$ and $G_1' \subseteq G_2'$; or
(ii) $G_1 \equiv \emptyset$.

Definition 19 (Union (Skeleton Location Graph))**.** The union of two skeleton location graphs G_1 and G_2, noted $G_1 \cup G_2$, is the least skeleton location graph G, unique up-to $\cdot \equiv \cdot$, such that $G_1 \subseteq G$ and $G_2 \subseteq G$.

Remark. The skeleton location graph resulting of the union of two skeleton location graphs is not necessarily the skeleton location graph of a location graph (there might exists roles that are not properly bound). ◁

Environment. Transitions of our semantics relate *environments* with other elements. Environments are formed of two components: (i) skeleton location graph; and (ii) a set of atoms.

Definition 20 (Environment)**.** An environment Γ is an element of $\mathbb{A} \times \mathbb{C}^s$. We write $\Gamma = \Delta \cdot G_s$. The set of all environments is noted $\mathbb{E}$.

Definition 21 (Elements of an environment)**.** Given an environment $\Gamma = \Delta \cdot G_s$, we define:

$$\Gamma.\texttt{names} \triangleq \Delta \qquad\qquad \Gamma.\texttt{graph} \triangleq G_s$$

Definition 22 (Environment union)**.** The union of two environments Γ_1 and Γ_2, noted $\Gamma_1 \cup \Gamma_2$, is defined as: $\Gamma_1.\texttt{names} \cup \Gamma_2.\texttt{names} \cdot \Gamma_1.\texttt{graph} \cup \Gamma_2.\texttt{graph}$.

Labels. The transitions of our semantics encompass labels. Labels expose two kinds of informations: (i) priority constraints, to temporarily block a transition depending on the neighbours of the location taking the transition; and (ii) interactions, which represent messages exchanged.

Remark. In this thesis, we do not use priority constraints. However, we still include them in this presentation for the sake of preciseness. ◁

The intuition behind an interaction, e.g. $r : a\langle V \rangle$, in a label of a transition t is that t can be fired if and only if there is, simultaneously, a location l which takes a transition with a label containing an interaction $\overline{r} : \overline{a}\langle V \rangle$. Essentially, it is an early binding message exchange.

Remark. Arbitrarily, we can say that the location taking the transition with the label $r : a\langle V \rangle$ *receives* the message and that the one with $\overline{r} : \overline{a}\langle V \rangle$ *sends* the message. Notice that, in such labels, the r or $\overline{r}$ refers to the role that is bound in the location (respectively in provided or required direction), regardless of the emission or reception of the message: a label $r : \overline{a}\langle V \rangle$ corresponds to the sending of a message on a role bound in the provided direction; and $\overline{r} : a\langle V \rangle$ corresponds to the reception of a message on a role bound in the required direction. ◁

The intuition behind a priority constraint, e.g. $r : a\langle V \rangle$, in a label of a transition t, is that t can be fired if and only if there is, in the rest of the graph, a location l binding $\overline{r}$ and which can take a transition with an interaction $\overline{r} : \hat{a}\langle V \rangle$ in the label. Similarly, a priority $r : \neg a\langle V \rangle$ enforce that there is no such location l. Since the location which takes the transition can bind r in two directions (r and $\overline{r}$), there are four variants of priorities.

Definition 23 (Priority constraint)**.** Given $\mathbb{R}$ a set of roles, $\mathbb{C}$ a set of channels and $\mathbb{V}$ a set of values, a priority constraint is an element of the set:

$$\mathbb{\Pi} = \{1, 2, 3, 4\} \times \mathbb{R} \times \mathbb{C} \times \mathbb{V}$$

Notation (Priority constraint)*.* We note:

(i) $r : a\langle V \rangle$ for $\langle 1, r, a, V \rangle$;
(ii) $\overline{r} : a\langle V \rangle$ for $\langle 2, r, a, V \rangle$;
(iii) $r : \neg a\langle V \rangle$ for $\langle 3, r, a, V \rangle$; and
(iv) $\overline{r} : \neg a\langle V \rangle$ for $\langle 4, r, a, V \rangle$.

In addition, we note $\hat{r} : a\langle V \rangle$ for either $r : a\langle V \rangle$ or $\overline{r} : a\langle V \rangle$[2]; and, given a $\hat{r} : a\langle V \rangle$, we note $\overline{\hat{r}} : a\langle V \rangle$ for the complement[3].

Notice that priority constraints are constraints on the fact that there exists a location that can (or can not) have transitions with messages on the given role, regardless of whether the message is sent or received; therefore, a priority constraint like $r : \overline{a}\langle V \rangle$ does not exist.

We define some accessors for elements of sets of priority constraints.

[2] $\hat{r} : a\langle V \rangle$ is a variable in $\{r : a\langle V \rangle, \overline{r} : a\langle V \rangle\}$.

[3] $\overline{\hat{r}} : a\langle V \rangle \triangleq \overline{r} : a\langle V \rangle$ if $\overline{r} : a\langle V \rangle = r : a\langle V \rangle$, and $r : a\langle V \rangle$ otherwise.

Definition 24 (Elements of priority constraints)**.** Let π be a set of priority constraints. We define:

$$\emptyset.\texttt{required} = \emptyset \qquad \{\overline{r} : a\langle V\rangle\}.\texttt{required} = \{r\} \qquad \{\overline{r} : \neg a\langle V\rangle\}.\texttt{required} = \{r\}$$

$$\{r : a\langle V\rangle\}.\texttt{required} = \emptyset \qquad \{r : \neg a\langle V\rangle\}.\texttt{required} = \emptyset$$

$$(\pi_1 \cup \pi_2).\texttt{required} = \pi_1.\texttt{required} \cup \pi_2.\texttt{required}$$

$$\emptyset.\texttt{provided} = \emptyset \qquad \{\overline{r} : a\langle V\rangle\}.\texttt{provided} = \emptyset \qquad \{\overline{r} : \neg a\langle V\rangle\}.\texttt{provided} = \emptyset$$

$$\{r : a\langle V\rangle\}.\texttt{provided} = \{r\} \qquad \{r : \neg a\langle V\rangle\}.\texttt{provided} = \{r\}$$

$$(\pi_1 \cup \pi_2).\texttt{provided} = \pi_1.\texttt{provided} \cup \pi_2.\texttt{provided}$$

$$\pi.\texttt{roles} \triangleq \pi.\texttt{provided} \cup \pi.\texttt{required}$$

Definition 25 (Interaction)**.** Given $\mathbb{R}$ a set of roles, $\mathbb{C}$ a set of channels and $\mathbb{V}$ a set of values, an interaction is an element of the set:

$$\mathbb{I} = \{1, 2, 3, 4\} \times \mathbb{R} \times \mathbb{C} \times \mathbb{V}$$

Notation (Interaction). We note:
(i) $r : a\langle V\rangle$ for $\langle 1, r, a, V\rangle$;
(ii) $\overline{r} : a\langle V\rangle$ for $\langle 2, r, a, V\rangle$;
(iii) $r : \overline{a}\langle V\rangle$ for $\langle 3, r, a, V\rangle$; and
(iv) $\overline{r} : \overline{a}\langle V\rangle$ for $\langle 4, r, a, V\rangle$.

We also define some accessors for the elements of sets of interactions.

Definition 26 (Elements of interactions)**.** Let ι be a set of interactions. Let $\iota.\texttt{required}$ be:

$$\emptyset.\texttt{required} = \emptyset \qquad \{\overline{r} : a\langle V\rangle\}.\texttt{required} = \{r\} \qquad \{\overline{r} : \overline{a}\langle V\rangle\}.\texttt{required} = \{r\}$$

$$\{r : a\langle V\rangle\}.\texttt{required} = \emptyset \qquad \{r : \overline{a}\langle V\rangle\}.\texttt{required} = \emptyset$$

$$(\iota_1 \cup \iota_2).\texttt{required} = \iota_1.\texttt{required} \cup \iota_2.\texttt{required}$$

$$\emptyset.\texttt{provided} = \emptyset \qquad \{\overline{r} : a\langle V\rangle\}.\texttt{provided} = \emptyset \qquad \{\overline{r} : \overline{a}\langle V\rangle\}.\texttt{provided} = \emptyset$$

$$\{r : a\langle V\rangle\}.\texttt{provided} = \{r\} \qquad \{r : \overline{a}\langle V\rangle\}.\texttt{provided} = \{r\}$$

$$(\iota_1 \cup \iota_2).\texttt{provided} = \iota_1.\texttt{provided} \cup \iota_2.\texttt{provided}$$

$$\iota.\texttt{roles} \triangleq \iota.\texttt{provided} \cup \iota.\texttt{required}$$

Finally, a label contains both priority constraints and interactions.

Definition 27 (Label)**.** A label is an element of $\mathcal{P}(\mathbb{\Pi}) \times \mathcal{P}(\mathbb{I})$.

Notice that $\mathbb{\Pi}$ and $\mathbb{I}$ insidiously hide $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{V}$. To make those appear more clearly, we define $\texttt{sLabel}(\cdot, \cdot, \cdot)$ which describes the set of label based on given basic sets.

Notation (Set of labels). The set of labels over $\mathbb{R}$, $\mathbb{C}$ and $\mathbb{V}$ is noted $\texttt{sLabel}(\mathbb{R}, \mathbb{C}, \mathbb{V})$. Said otherwise, we note $\texttt{sLabel}(\mathbb{R}, \mathbb{C}, \mathbb{V})$ for $\{\{1, 2, 3, 4\} \times \mathbb{R} \times \mathbb{C} \times \mathbb{V}\}^2$

Definition 28 (Elements of a label)**.** Given a label $\Lambda = \langle \pi, \iota \rangle$, we define:

$$\Lambda.\texttt{prior} = \pi \qquad \Lambda.\texttt{sync} = \iota \qquad \Lambda.\texttt{roles} \triangleq \Lambda.\texttt{sync.roles} \cup \Lambda.\texttt{prior.roles}$$

Unconstrained location transitions. The starting point of location graph semantics is the definition of a set $\mathcal{T}_u$ of unconstrained location transitions. These transitions describe the behaviour of individual locations.

Definition 29 (Unconstrained location transition)**.** An unconstrained transition is a quadruplet (noted $\Gamma \rhd L \xrightarrow{\Lambda} C$) composed of the following elements:

1. An environment $\Gamma = \Delta \cdot \emptyset$;
2. A location $L \in \mathbb{L}$ such that $\texttt{supp}(L) \subseteq \Delta$;
3. A label $\Lambda = \langle \pi, \iota \rangle$, such that (i) $\pi.\texttt{required} \subseteq L.\texttt{required}$; (ii) $\pi.\texttt{provided} \subseteq L.\texttt{provided}$; (iii) $\iota.\texttt{required} \subseteq L.\texttt{required}$; and (iv) $\iota.\texttt{provided} \subseteq L.\texttt{provided}$;
4. A location graph $C \in \mathbb{G}$.

In addition, we require $\mathcal{T}_u$ to include special transitions:

$$\forall L \in \mathbb{L}, r \in L.\texttt{roles}, \Delta \cdot \texttt{supp}(L) \subseteq \Delta \Rightarrow \Delta \cdot \emptyset \rhd L \xrightarrow{\langle \epsilon, \{r : \overline{\texttt{rmv}} L\} \rangle} \emptyset \in \mathcal{T}_u$$

Such transitions make public (on the special channel `rmv`) the fact that a location L is removed.

Notice that the locations, location graphs and the skeletons in the transition are independents of the channels and the values (except for messages on `rmv`, which contain locations). Let $\texttt{trans}(\mathbb{G}, \mathbb{\Lambda})$ be the set of all unconstrained location transitions sets (all $\mathcal{T}_u$) over graphs in $\mathbb{G}$ with labels in $\mathbb{\Lambda}$.

Authorisation function. An important aspect of the location graph framework is the *authorisation function* of a graph, used for graph transitions (see below). An authorisation function is used to authorise or forbid unconstrained location transitions.

Definition 30 (Authorisation function)**.** An *authorisation function* is a predicate over skeleton graphs and unconstrained location transitions.

Notation (Authorisation function). Authorisation functions are noted $\texttt{Auth}(G_s, t)$ (and decorated variants) where $G_s \in \mathbb{G}_s$ and $t \in \mathcal{T}_u$.

Let $\texttt{sAuth}(\mathbb{G}, \mathcal{T}_u)$ be the set of all authorisation functions over $\mathbb{G}_s$ and $\mathcal{T}_u$.

Definition 31 (Trivial authorisation function)**.** We call *trivial* the authorisation function which always holds.

Graph transitions. We now define the semantics of location graphs. We first define a rule that lift unconstrained location transitions to transitions for individual locations (rule (Trans)), then we define the rules of composed graphs (rules (Comp) and (Ctx)).

The semantics of location graphs are expressed as a transition system where transitions have the form $\Gamma \vdash_{\mathcal{T}_u} G_1 \xrightarrow{\Lambda} G_2$.

The first step is to define a few relations used hereafter in the reduction rules.

First, a priority is satisfied if there is (or is not) a location that can take a transition with an adequate interaction.

Definition 32 (Priority satisfaction)**.** A location graph G, in an environment Γ, satisfies a priority $\rho = r : a\langle V \rangle \in \mathbb{\Pi}$ (noted $G \models_\Gamma \rho$) if and only if:

$$\begin{aligned} G \models_\Gamma \rho \triangleq\ & \Sigma(G) \subseteq \Gamma.\texttt{graph} \\ & \wedge\, r \in G.\texttt{unbound} \\ & \wedge\, \exists G' \in \mathbb{G} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \wedge \overline{r} : \overline{a}\langle V \rangle \in \Lambda.\texttt{sync} \end{aligned}$$

Likewise, if $\rho = \overline{r} : a\langle V \rangle$:

$$\begin{aligned} G \models_\Gamma \rho \triangleq\ & \Sigma(G) \subseteq \Gamma.\texttt{graph} \\ & \wedge\, r \in G.\texttt{unbound} \\ & \wedge\, \exists G' \in \mathbb{G} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \wedge r : \overline{a}\langle V \rangle \in \Lambda.\texttt{sync} \end{aligned}$$

If $\rho = r : \neg a\langle V\rangle$, L satisfies ρ if and only if:

$$\begin{aligned} G \models_\Gamma \rho \triangleq\ & \Sigma(G) \subseteq \Gamma.\texttt{graph} \\ & \wedge\ r \in G.\texttt{unbound} \\ & \wedge\ \neg\exists G' \in \mathbb{G} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \wedge \overline{r} : \overline{a}\langle V\rangle \in \Lambda.\texttt{sync} \end{aligned}$$

And, likewise, if $\rho = \overline{r} : \neg a\langle V\rangle$:

$$\begin{aligned} G \models_\Gamma \rho \triangleq\ & \Sigma(G) \subseteq \Gamma.\texttt{graph} \\ & \wedge\ r \in G.\texttt{unbound} \\ & \wedge\ \neg\exists G' \in \mathbb{G} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \wedge r : \overline{a}\langle V\rangle \in \Lambda.\texttt{sync} \end{aligned}$$

We want to state that all priority constraints which concern bound roles are enforced. In particular, when composing two graphs G_1 and G_2 which take a transition simultaneously, we want to extract priorities in the label of G_1 which concerns G_2, and verify that G_2 satisfies them. The predicate $\texttt{Cond}_P(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot)$ states does that.

Definition 33 (Priority satisfaction of composed graphs)**.**

$$\begin{aligned} \texttt{Cond}_P(s,\pi,\pi_1,\pi_2,\Gamma,G_1,G_2) \triangleq\ & \pi = \{\rho \in \pi_1 \cup \pi_2 | \rho.\texttt{roles} \in (G_1 \parallel G_2).\texttt{unbound}\} \\ & \wedge \bigwedge_{\rho \in \pi_1 \setminus \pi} G_2 \models_\Gamma \rho \\ & \wedge \bigwedge_{\rho \in \pi_2 \setminus \pi} G_1 \models_\Gamma \rho \end{aligned}$$

Similarly, for interactions, we define a function $\texttt{seval}(\cdot)$ which removes matching interactions of a set of interactions, and we then define a predicate $\texttt{Cond}_I(\cdot,\cdot,\cdot,\cdot)$ which asserts that interactions that should be matched are matched.

Definition 34 (Evaluation of matching interactions)**.**

$$\texttt{seval}(\sigma) \triangleq \begin{cases} \texttt{seval}(\sigma') & \text{if } \sigma = (\sigma' \cup \{r : a\langle V\rangle, \overline{r} : \overline{a}\langle V\rangle\}) \wedge \emptyset = (\sigma' \cap \{r : a\langle V\rangle, \overline{r} : \overline{a}\langle V\rangle\}) \\ \texttt{seval}(\sigma') & \text{if } \sigma = (\sigma' \cup \{r : \overline{a}\langle V\rangle, \overline{r} : a\langle V\rangle\}) \wedge \emptyset = (\sigma' \cap \{r : \overline{a}\langle V\rangle, \overline{r} : a\langle V\rangle\}) \\ \sigma & \text{otherwise} \end{cases}$$

The predicate $\texttt{Cond}_I(\cdot,\cdot,\cdot,\cdot)$ states that interactions are either matched; or on unbound roles.

Definition 35 (Correct matching of a union of interaction sets)**.** $\texttt{Cond}_I(\iota,\iota_1,\iota_2,G) \triangleq \iota = \texttt{seval}(\iota_1 \cup \iota_2) \wedge \iota.\texttt{roles} \subseteq G.\texttt{unbound}$

The predicate $\texttt{Cond}(\cdot,\cdot)$ states that the graph should be taken into account in the environment.

Definition 36 (Correct environment)**.** $\texttt{Cond}(\Gamma, G) \triangleq \texttt{supp}(G) \subseteq \Gamma.\texttt{names} \wedge \Sigma(G) \subseteq \Gamma.\texttt{graph}$

We also define $\texttt{Ind}_P(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot)$ and $\texttt{Ind}_I(\cdot,\cdot)$ which are analogous, but for when only one side of the graph reduces.

Definition 37 (Priority satisfaction of composed graphs (2))**.**

$$\begin{aligned} \texttt{Ind}_P(s,\pi,\varpi,\Gamma,G,E) \triangleq\ & \pi = \{\rho \in \varpi | \rho.\texttt{roles} \in (G \parallel E).\texttt{unbound}\} \\ & \wedge \bigwedge_{\rho \in \varpi \setminus \pi} E \models_\Gamma \rho \end{aligned}$$

Definition 38 (Independent interaction set)**.** $\texttt{Ind}_I(\iota, G) \triangleq \iota.\texttt{roles} \subseteq G.\texttt{unbound}$

$$(\textsc{Trans})\ \frac{\Gamma.\texttt{names} \cdot \emptyset \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_u \qquad \Sigma(L) \in \Gamma.\texttt{graph} \qquad \texttt{Auth}(\Gamma.\texttt{graph}, \Gamma.\texttt{names} \cdot \emptyset \rhd L \xrightarrow{\Lambda} C)}{\Gamma \vdash_{\mathcal{T}_u} L \xrightarrow{\Lambda} C}$$

$$(\textsc{Comp})\ \frac{\begin{array}{c}\Gamma \vdash_{\mathcal{T}_u} G_1 \xrightarrow{\langle \pi_1, \iota_1 \rangle} G_1' \qquad \Gamma \vdash_{\mathcal{T}_u} G_2 \xrightarrow{\langle \pi_2, \iota_2 \rangle} G_2' \\ \texttt{Cond}_P(s, \pi, \pi_1, \pi_2, \Gamma, G_1, G_2) \qquad \texttt{Cond}_I(\iota, \iota_1, \iota_2, G_1 \parallel G_2) \qquad \texttt{Cond}(\Gamma, G_1 \parallel G_2)\end{array}}{\Gamma \vdash_{\mathcal{T}_u} G_1 \parallel G_2 \xrightarrow{\langle \pi, \iota \rangle} G_1' \parallel G_2'}$$

$$(\textsc{Ctx})\ \frac{\Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\langle \varpi, \iota \rangle} G' \qquad \texttt{Ind}_P(s, \pi, \varpi, \Gamma, G, E) \qquad \texttt{Ind}_I(\iota, G \parallel E) \qquad \texttt{Cond}(\Gamma, G \parallel E)}{\Gamma \vdash_{\mathcal{T}_u} G \parallel E \xrightarrow{\langle \pi, \iota \rangle} G' \parallel E}$$

Priority constraints of unbound roles always hold (see rule (Comp) and Definition 33). Thus it is possible to allow a transition if a role r is unbound having a priority constraint π for the rule that includes both $\overline{r} : a\langle\star\rangle$ and $\overline{r} : \neg a\langle\star\rangle$.

We show that if a graph takes a transition, then the roles that appear in the interactions are necessarily unbound. Intuitively, to take the transition, all interactions on bound roles should be matched, and therefore removed by $\texttt{seval}(\cdot)$.

Lemma 4.

$$\forall G \in \mathbb{G} \cdot \Delta \cdot G_s \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \wedge \hat{r} : \hat{a}\langle V \rangle \in \Lambda.\texttt{sync} \Rightarrow r \in G.\texttt{unbound}$$

Proof. By induction on G:

Case $G = \emptyset$: Only the rule (Trans) can apply. From the premisses of (Trans), $\Delta \cdot \emptyset \rhd G \xrightarrow{\Lambda} G'$.
From the conditions on unconstrained location transitions, $\Lambda.\texttt{sync.provided} \subseteq G.\texttt{prov}$ and $\Lambda.\texttt{sync.required} \subseteq G.\texttt{req}$.
Yet, since $G = \emptyset$, from Definition 8, $G.\texttt{prov} = \emptyset$ and $G.\texttt{req} = \emptyset$. Therefore, $\Lambda.\texttt{sync.provided} = \emptyset$ and $\Lambda.\texttt{sync.required} = \emptyset$.
Therefore, $\Lambda.\texttt{sync} = \emptyset$ and $\hat{r} : \hat{a}\langle V \rangle \in \Lambda.\texttt{sync}$ is a vacuous statement. Therefore, the result holds.

Case $G = [P : s \lhd p \bullet r]$: Only the rule (Trans) can apply. From the premisses of (Trans), $\Delta \cdot \emptyset \rhd G \xrightarrow{\Lambda} G'$.
From Definition 8, $G.\texttt{unbound} = G.\texttt{roles} \setminus G.\texttt{bound} = (G.\texttt{prov} \cup G.\texttt{req}) \setminus (G.\texttt{prov} \cap G.\texttt{req})$.
Since $G = [P : s \lhd p \bullet r]$, from Definition 6, $\texttt{WF}([P : s \lhd p \bullet r])$ holds and, from Definition 8, $G.\texttt{prov} = [P : s \lhd p \bullet r].\texttt{provided}$ and $G.\texttt{req} = [P : s \lhd p \bullet r].\texttt{required}$.
By Definition 5, $\texttt{WF}([P : s \lhd p \bullet r]) \Leftrightarrow [P : s \lhd p \bullet r].\texttt{provided} \cap [P : s \lhd p \bullet r].\texttt{required} = \emptyset$.
Therefore, $G.\texttt{unbound} = G.\texttt{prov} \cup G.\texttt{req}$.
In addition, from the conditions on unconstrained location transitions, $\Lambda.\texttt{sync.provided} \subseteq G.\texttt{prov} \subseteq G.\texttt{unbound}$ and $\Lambda.\texttt{sync.required} \subseteq G.\texttt{req} \subseteq G.\texttt{unbound}$.
Finally, for each $\hat{r} : \hat{a}\langle V \rangle \in \Lambda.\texttt{sync}$, $r \in \Lambda.\texttt{sync.provided}$ or $r \in \Lambda.\texttt{sync.required}$. In both cases, $r \in G.\texttt{unbound}$.

Case $G = G_1 \parallel G_2$: Two rules can apply: (Ctx) and (Comp).

Case (Ctx): By symmetry, suppose that G_1 is the subgraph reducing: $\Delta \cdot G_s \vdash_{\mathcal{T}_u} G_1 \xrightarrow{\Lambda_1} G_1'$ and $G' = G_1' \parallel G_2$.
From the premisses of (Ctx), $\texttt{Ind}_I(\Lambda.\texttt{sync}, G_1 \parallel G_2)$.
By definition of $\texttt{Ind}_I(\sigma, G)$, $\Lambda.\texttt{sync.roles} \subseteq G.\texttt{unbound}$.

Case (Comp): From the premisses of (Comp), $\texttt{Cond}_I(\Lambda.\texttt{sync}, \sigma_1, \sigma_2, G)$. By definition of $\texttt{Cond}_I(\sigma, \sigma_1, \sigma_2, G)$, $\Lambda.\texttt{sync.roles} \subseteq G.\texttt{unbound}$.

□

Example 3 (Ping-pong locations)**.** We consider a very simple example, in which two locations exchange a token $\star$.
For this example, our processes can have two forms: one to indicate the location expects to receive the $\star$, and the other to indicate the location attempts to send the $\star$. Therefore, we simply choose $\mathbb{P} = \{\top, \bot\}$. We do not need sorts in this example, therefore we consider that $\mathbb{S} = \{\_\}$. We need a single role r, thus we set $\mathbb{R} = \{r\}$. Similarly, we only use a single channel, so we define $\mathbb{C} = \{a, \texttt{rmv}\}$ (notice that, to fulfil the definition, we have to include `rmv` nonetheless). Finally, of course our unique value is $\star$, therefore $\mathbb{V} = \{\star\}$.
An example of location, using these base sets, is $[\top : \_ \lhd \{r\} \bullet \emptyset]$. An example of location graph is $[\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}]$.
However, for the sake of the example, notice that $[\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \{r\} \bullet \emptyset]$, in which both locations bind r in the same direction, is a location pregraph, but not a location graph, since $\texttt{WF}_G([\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \{r\} \bullet \emptyset])$ holds only if $\texttt{separate}([\top : \_ \lhd \{r\} \bullet \emptyset], [\bot : \_ \lhd \{r\} \bullet \emptyset])$ holds, which itself requires that $[\top : \_ \lhd \{r\} \bullet \emptyset].\texttt{prov} \cap [\bot : \_ \lhd \{r\} \bullet \emptyset].\texttt{prov} = \{r\} = \emptyset$, which, of course, does not hold.
Our set of unconstrained location transitions, which we note $\mathcal{T}_p$ in this example, can be defined as follow:

$$\begin{aligned}
\mathcal{T}_p = \{&\Delta \cdot \emptyset \rhd [\top : _ \lhd \{r\} \bullet \emptyset] \xrightarrow{\langle \emptyset, r : a\langle\star\rangle\rangle} [\bot : _ \lhd \{r\} \bullet \emptyset],\\
&\Delta \cdot \emptyset \rhd [\bot : _ \lhd \{r\} \bullet \emptyset] \xrightarrow{\langle \emptyset, r : \overline{a}\langle\star\rangle\rangle} [\top : _ \lhd \{r\} \bullet \emptyset],\\
&\Delta \cdot \emptyset \rhd [\top : _ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \overline{r} : a\langle\star\rangle\rangle} [\bot : _ \lhd \emptyset \bullet \{r\}],\\
&\Delta \cdot \emptyset \rhd [\bot : _ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \overline{r} : \overline{a}\langle\star\rangle\rangle} [\top : _ \lhd \emptyset \bullet \{r\}]\}
\end{aligned}$$

And, for our authorisation function, we take the trivial one function $\mathcal{A}$ which always holds.
If we consider again the location graph $G = [\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}]$ we see that it can reduce to $G' = [\bot : \_ \lhd \{r\} \bullet \emptyset] \parallel [\top : \_ \lhd \emptyset \bullet \{r\}]$, with a $\star$ exchanged (the derivation tree is given at the end of this example).
We note G_s for $\Sigma(G) = \Sigma([\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}]) = [\_ \lhd \{r\} \bullet \emptyset] \parallel [\_ \lhd \emptyset \bullet \{r\}]$
Of course, $\texttt{Cond}_P(s, \emptyset, \emptyset, \emptyset, \Delta \cdot G_s, [\top : \_ \lhd \{r\} \bullet \emptyset], [\bot : \_ \lhd \emptyset \bullet \{r\}])$ vacuously holds. Concerning $\texttt{Cond}_I(\emptyset, \{r : a\langle\star\rangle\}, \{\overline{r} : \overline{a}\langle\star\rangle\}, [\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}])$, by definition, $\texttt{seval}(\{\overline{r} : \overline{a}\langle\star\rangle\} \cup \{r : a\langle\star\rangle\}) = \emptyset$, and $\emptyset.\texttt{roles} = \emptyset$, and therefore the predicate holds. Finally, by choosing $\Delta \triangleq \mathbb{R} \cup \mathbb{C}$, we have that $\texttt{Cond}(\Delta \cdot G_s, G)$ holds.

$$(\textsc{Trans})\ \frac{\Delta \cdot G_s \rhd [\top : \_ \lhd \{r\} \bullet \emptyset] \xrightarrow{\langle \emptyset, \{r : a\langle\star\rangle\}\rangle} [\bot : \_ \lhd \{r\} \bullet \emptyset] \in \mathcal{T}_p \quad \Sigma([\top : \_ \lhd \{r\} \bullet \emptyset]) \in G_s \quad \mathcal{A}(G_s, \Delta \cdot G_s \rhd [\top : \_ \lhd \{r\} \bullet \emptyset] \xrightarrow{\langle \emptyset, \{r : a\langle\star\rangle\}\rangle} [\bot : \_ \lhd \{r\} \bullet \emptyset])}{\Delta \cdot G_s \vdash_{\mathcal{T}_p} [\top : \_ \lhd \{r\} \bullet \emptyset] \xrightarrow{\langle \emptyset, \{r : a\langle\star\rangle\}\rangle} [\bot : \_ \lhd \{r\} \bullet \emptyset]}$$

$$(\textsc{Trans})\ \frac{\Delta \cdot G_s \rhd [\bot : \_ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \{\overline{r} : \overline{a}\langle\star\rangle\}\rangle} [\bot : \_ \lhd \emptyset \bullet \{r\}] \in \mathcal{T}_p \quad \Sigma([\bot : \_ \lhd \emptyset \bullet \{r\}]) \in G_s \quad \mathcal{A}(G_s, \Delta \cdot G_s \rhd [\bot : \_ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \{\overline{r} : \overline{a}\langle\star\rangle\}\rangle} [\bot : \_ \lhd \emptyset \bullet \{r\}])}{\Delta \cdot G_s \vdash_{\mathcal{T}_p} [\bot : \_ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \{\overline{r} : \overline{a}\langle\star\rangle\}\rangle} [\bot : \_ \lhd \emptyset \bullet \{r\}]}$$

$$(\textsc{Comp})\ \frac{\begin{array}{c}\texttt{Cond}_P(s, \emptyset, \emptyset, \emptyset, \Delta \cdot G_s, [\top : \_ \lhd \{r\} \bullet \emptyset], [\bot : \_ \lhd \emptyset \bullet \{r\}]) \\ \texttt{Cond}_I(\emptyset, \{r : a\langle\star\rangle\}, \{\overline{r} : \overline{a}\langle\star\rangle\}, [\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}]) \qquad \texttt{Cond}(\Delta \cdot G_s, [\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}])\end{array}}{\Delta \cdot G_s \vdash_{\mathcal{T}_p} [\top : \_ \lhd \{r\} \bullet \emptyset] \parallel [\bot : \_ \lhd \emptyset \bullet \{r\}] \xrightarrow{\langle \emptyset, \emptyset \rangle} [\bot : \_ \lhd \{r\} \bullet \emptyset] \parallel [\top : \_ \lhd \emptyset \bullet \{r\}]}$$

3.1.3 Additional operations

We conclude this section with the definition of additional functions and lemmas that are not part of the original draft [33], but which proved useful.

Size of a location graph. Our first addition is to formally define the size of a location graph, which follows the intuition and counts, by structural induction, the number of locations in the graph.

Definition 39 (Location graph size)**.** The size of a location graph is define inductively as follow:

$$\text{size}(\emptyset) \triangleq 0 \qquad \text{size}(L) \triangleq 1 \qquad \text{size}(G_1 \parallel G_2) \triangleq \text{size}(G_1) + \text{size}(G_2)$$

Of course, this notion of size is consistent with our structural equivalence, i.e. two equivalent graphs have the same size.

Lemma 5.

$$\forall G_1, G_2 \in \mathbb{G} \cdot G_1 \equiv G_2 \Rightarrow \text{size}(G_1) = \text{size}(G_2)$$

Proof. By induction on the rules of structural equivalence:
$G_1 = \emptyset \parallel G_2 \equiv G_2$: $\text{size}(G_1) = \text{size}(\emptyset) + \text{size}(G_2) = 0 + \text{size}(G_2) = \text{size}(G_2)$

$G_1 = G_a \parallel G_b$ AND $G_2 = G_b \parallel G_a$: $\text{size}(G_1) = \text{size}(G_a) + \text{size}(G_b) = \text{size}(G_2)$

$G_1 = G_a \parallel (G_b \parallel G_c)$ AND $G_2 = (G_a \parallel G_b) \parallel G_c$: $\text{size}(G_1) = \text{size}(G_a) + (\text{size}(G_b) + \text{size}(G_c)) = (\text{size}(G_a) + \text{size}(G_b)) + \text{size}(G_c) = \text{size}(G_2)$

$G_1 = G_a \parallel G$, $G_2 = G_b \parallel G$ AND $G_a \equiv G_b$: From the induction hypothesis, $\text{size}(G_a) = \text{size}(G_b)$. Therefore, $\text{size}(G_1) = \text{size}(G_a) + \text{size}(G) = \text{size}(G_b) + \text{size}(G) = \text{size}(G_2)$

□

We also prove a small additional lemma which allows us to deduce that a graph is empty or contains a single location when its size is 0 or 1.

Lemma 6. $\forall G \in \mathbb{G} \cdot \text{size}(G) = 0 \Leftrightarrow G \equiv \emptyset$ *and* $\forall G \in \mathbb{G} \cdot \text{size}(G) = 1 \Leftrightarrow (\exists L \in \mathbb{L} \cdot G \equiv L)$.

Proof. We analyse separately the two statements.
STATEMENT $\text{size}(G) = 0 \Leftrightarrow G \equiv \emptyset$: We prove separately the cases $\Rightarrow$ and $\Leftarrow$:

- $\Rightarrow$: By induction on G:
  - CASE $G = \emptyset$: Direct, by reflexivity of $\equiv$.
  - CASE $G = L \in \mathbb{L}$: By hypothesis, $\text{size}(G) = 0$, and, $G = L \in \mathbb{L}$. From Definition 39, $\text{size}(L) = 1$.
    Therefore $\text{size}(L) = 0 = 1$. Contradiction.
  - CASE $G = G_1 \parallel G_2$: $\text{size}(G_1 \parallel G_2) = \text{size}(G_1) + \text{size}(G_2) = 0$. Therefore, $\text{size}(G_1) = \text{size}(G_2) = 0$. From the induction hypothesis, $G_1 \equiv G_2 \equiv \emptyset$. Finally, $G = G_1 \parallel G_2 \equiv \emptyset \parallel \emptyset \equiv \emptyset$.
- $\Leftarrow$: Follows directly from Lemma 5.

STATEMENT $\text{size}(G) = 1 \Leftrightarrow (\exists L \in \mathbb{L} \cdot G \equiv L)$: We prove separately the cases $\Rightarrow$ and $\Leftarrow$:

- $\Rightarrow$: By induction on G:
  - CASE $G = \emptyset$: $1 = \text{size}(G) = \text{size}(\emptyset) = 0$. Contradiction.
  - CASE $G = L' \in \mathbb{L}$: Trivial, $L = L'$.
  - CASE $G = G_1 \parallel G_2$: $\text{size}(G_1 \parallel G_2) = \text{size}(G_1) + \text{size}(G_2) = 1$. Therefore, either (i) $\text{size}(G_1) = 0$ and $\text{size}(G_2) = 1$; or $\text{size}(G_1) = 1$ and $\text{size}(G_2) = 0$.
    By symmetry, we consider that $\text{size}(G_1) = 0$ and $\text{size}(G_2) = 1$. From the induction hypothesis, $G_1 \equiv \emptyset$ and $\exists L \in \mathbb{L} \cdot G_2 \equiv L$. Therefore, $\exists L \in \mathbb{L} \cdot G = G_1 \parallel G_2 \equiv \emptyset \parallel L \equiv L$.
- $\Leftarrow$: Direct.

□

Multiset of locations. Our second helper function maps a location graph to a multiset of its locations.

Definition 40 (Locations of a graph)**.** The multiset of locations of a location graphs is defined as follow:

$$\text{locations}(\emptyset) \triangleq \emptyset \qquad \text{locations}(L) \triangleq \{\!\{L\}\!\} \qquad \text{locations}(G_1 \parallel G_2) \triangleq \text{locations}(G_1) \cup \text{locations}(G_2)$$

Location graph inclusion. Finally, we introduce a notion of graph inclusion. Again, it follows the intuition: a graph is included in an other if the other contains all locations of the first.

Definition 41 (Inclusion)**.** A location graph G_1 is included in G_2 (noted $G_1 \subseteq G_2$) if and only if:
(i) If $G_1 \equiv L \parallel G_1'$, then $G_2 \equiv L \parallel G_2'$ and $G_1' \subseteq G_2'$; or
(ii) $G_1 \equiv \emptyset$.

With this definition comes some lemmas. The first states that the subgraphs of a graph are included in the graph.

Lemma 7.

$$\forall G, G_1, G_2 \cdot G_1 \parallel G_2 \equiv G \Rightarrow G_1 \subseteq G$$

Proof. By induction on $\text{size}(G_1)$:

Case $\text{size}(G_1) = 0$: From Lemma 6, $G_1 = \emptyset$. The result is then direct, from case (ii) of the definition.

Case $\text{size}(G_1) = 1$: From Lemma 6, $G_1 = L$ for some location L. The result is then direct, from case (i) of the definition.

Case $\text{size}(G_1) = n + 1$ $(n \geq 1)$:

Induction Hypothesis. $\forall G_1' \cdot \text{size}(G_1') \leq n \Rightarrow \forall G' \cdot G_1' \parallel G_2 \equiv G' \Rightarrow G_1' \subseteq G'$.
Since $\text{size}(G_1) = n + 1$, with $n \geq 1$, then from Lemma 6, $G_1 \equiv L \parallel G_1'$, for some location L and some location graph G_1'. Therefore, from the case (i) of the definition, we have to prove that there exists G' such that: (i) $G \equiv L \parallel G'$; and (ii) $G_1' \subseteq G'$.
From the hypothesis, since $G_1 \parallel G_2 \equiv G$, then $(L \parallel G_1') \parallel G_2 \equiv G$. Therefore, from the rules of structural equivalence: $L \parallel (G_1' \parallel G_2) \equiv G$. We therefore take $G' = G_1' \parallel G_2$, which conclude the first point.
From Definition 39, $\text{size}(G_1) = n+1 = \text{size}(L)+\text{size}(G_1') = 1+\text{size}(G_1')$, therefore $\text{size}(G_1') = n$. Therefore, from the induction hypothesis: $\forall G' \cdot G_1' \parallel G_2 \equiv G' \Rightarrow G_1' \subseteq G'$. In particular, this applies to the G' we chose above, therefore $G_1' \subseteq G'$, which conclude the second point. □

The second states that the size of a graph included in an other is at most the size of the other.

Lemma 8. $\forall G_1, G_2 \cdot G_1 \subseteq G_2 \Rightarrow \text{size}(G_1) \leq \text{size}(G_2)$

Proof. Direct, by induction on G_1. □

Finally, we show that if a graph is included in an other and both have the same size, then they are equivalent.

Lemma 9.

$$\forall G_1, G_2 \cdot G_1 \subseteq G_2 \wedge \text{size}(G_1) = \text{size}(G_2) \Rightarrow G_1 \equiv G_2$$

Proof. By induction on G_1:

Case $G_1 = \emptyset$: By definition of $\text{size}([\cdot])$, $\text{size}(G_1) = 0$. By hypothesis, $\text{size}(G_1) = \text{size}(G_2) = 0$.
From Lemma 6, $G_2 \equiv \emptyset$, and, by transitivity of $\equiv$, $G_2 \equiv G_1$.

Case $G_1 = L \in \mathbb{L}$: By definition of $\text{size}([\cdot])$, $\text{size}(G_1) = 1$. By hypothesis, $\text{size}(G_1) = \text{size}(G_2) = 1$.
From Lemma 6, $G_2 \equiv L_2 \equiv L_2 \parallel \emptyset$ for some location L_2.
By definition of graph inclusion (Definition 41, case (i)), since $G_1 \equiv L \parallel \emptyset$, we have $G_2 \equiv L \parallel G_2'$ (with $\emptyset \subseteq G_2'$).

In conclusion, we have $G_2 \equiv L_2 \parallel \emptyset \equiv L \parallel G_2'$, therefore $G_2 \equiv L \equiv G_1$.

Case $G_1 = G_{1a} \parallel G_{1b}$: If $G_{1a} \equiv \emptyset$ and $G_{1b} \equiv \emptyset$, the result holds directly from the induction hypothesis.
Otherwise, $G_1 \equiv L \parallel G_1'$, for L a location from either G_{1a} or G_{1b}.
By hypothesis, $G_1 \subseteq G_2$. By definition of $\subseteq$, since $G_1 \equiv L \parallel G_1'$, $G_2 \equiv L \parallel G_2'$ and $G_1' \subseteq G_2'$.
By definition of size$(\cdot)$, size$(G_1) = 1 + \text{size}(G_1')$ and size$(G_2) = 1 + \text{size}(G_2')$. By hypothesis, size$(G_1) =$ size(G_2). Therefore, size$(G_1') =$ size(G_2').
From our induction hypothesis, $G_1' \equiv G_2'$.
Therefore, $G_1 \equiv L \parallel G_1' \equiv L \parallel G_2' \equiv G_2$.

□

Location invariant. We show a useful helper lemma which states, informally, that if a property on locations is preserved under unconstrained location transitions, then it is also preserved under full graph reduction.

Remark. This lemma is the only one in this whole work which uses second order logic. ◁

Lemma 10 (Location invariant)**.** *For any predicate $\mathcal{P}(L)$ on locations, if*

$$\forall \mathcal{T}_u \cdot \forall \Gamma \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_u \cdot \mathcal{P}(L) \Rightarrow (\forall L' \in C \cdot \mathcal{P}(L'))$$

then

$$\forall G, G' \in \mathbb{G} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \Rightarrow (\forall L \in G \cdot \mathcal{P}(L)) \Rightarrow (\forall L \in G' \cdot \mathcal{P}(L))$$

Proof. By induction on the graph G:
Case $G = \emptyset$: Vacuously holds.

Case $G = L = [P : s \lhd p \bullet r]$: Only the rule (Trans) can apply. From the premisses of the rule, $\Gamma \rhd L \xrightarrow{\Lambda} G' \in \mathcal{T}_u$.
By hypothesis, $\forall \Gamma \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_u \cdot \mathcal{P}(L) \Rightarrow (\forall L' \in C \cdot \mathcal{P}(L'))$ and $\forall L \in G \cdot \mathcal{P}(L)$.
Therefore, since $G \equiv L$, $\forall L' \in G' \cdot \mathcal{P}(L')$.

Case $G = G_1 \parallel G_2$: G either takes a (Comp) transition or a (Ctx) transition. In both cases, the reasoning is the same. We detail the case (Comp).
By hypothesis, $\forall L \in G \cdot \mathcal{P}(L)$, therefore, since $G = G_1 \parallel G_2$, $\forall L \in G_1 \cdot \mathcal{P}(L)$ and $\forall L \in G_2 \cdot \mathcal{P}(L)$.
By definition of (Comp): $\Gamma \vdash_{\mathcal{T}_u} G_1 \xrightarrow{\Lambda_1} G_1'$, $\Gamma \vdash_{\mathcal{T}_u} G_2 \xrightarrow{\Lambda_2} G_2'$, and $G' = G_1' \parallel G_2'$.
By induction hypothesis, $\forall L \in G_1' \cdot \mathcal{P}(L)$ and $\forall L \in G_2' \cdot \mathcal{P}(L)$.
Therefore, $\forall L \in G_1' \parallel G_2' \cdot \mathcal{P}(L)$, i.e. $\forall L \in G' \cdot \mathcal{P}(L)$.

□

3.2 Comparing Location Graphs

As in process calculi, we are interested in comparing location graphs. Such comparison is appealing for various purposes, for instance to develop programs in a modular way: if we are able to prove that a module behave the same way as an other, one can replace the other in a bigger system, without changing the global behaviour of the system. An other example (which is our main motivation in this thesis) is that it can simplify the analysis of a location graph: if some properties are shown on a particular instance, an adequate comparison tool would directly prove that property for all equivalent instances.

This is traditionally done using an adequate relation of (bi)simulation between systems. Simulations and bisimulations are a class of relation of prime importance in the domain of process calculi (see for instance [51, 53], the most complete work on simulations). The initial presentation paper of the location graph framework [57, 33] introduced a notion of simulation for location graphs. However, this initial notion was defined as a relation on location graphs belonging to the same model (i.e. the simulation relation is a subset of $\mathbb{G} \times \mathbb{G}$, for a given $\mathbb{G}$). We can already anticipate that such constraint is problematic for our goal: we work on encapsulation and we will attempt to relate a graph and its encapsulated

counterpart: both will not be defined using the same basic sets[4]. We therefore need a way to compare *heterogeneous* instances of location graphs, which is the goal of this section.

We propose a method to overcome this issue, which consists in a (conservative) extension of location graph simulations, in which simulations are relations over $\mathbb{G}_1 \times \mathbb{G}_2$, for two location graph models $\mathbb{G}_1$ and $\mathbb{G}_2$.

> We could be tempted to take an other approach in which we would define the *product* of location graphs, allowing us to create a third model $\mathbb{G}$, which embeds both models $\mathbb{G}_1$ and $\mathbb{G}_2$, with the hope to fall back on the original definition of simulation. We explored that possibility, which we saw the early developments in Appendix B.
> Unfortunately, this approach is less trivial than what one could expect at first, as we show in the appendix, and we could not find an adequate result. This failure motivates an analysis of the location graph framework, from a categoric viewpoint, with a sound notion of product.

In the first subsection, we recall the original definition of a simulation relation for location graphs. Then, we introduce our new definition. Finally, we extend a bit further by presenting *partial bisimulations*, directly inspired by Rutten (in [49]), which is a slight modification allowing to ignore some labels of the simulations.

3.2.1 Simulation relations for location graphs

Remark. The content of this section (Definitions 42, 43, and 44) was previously introduced in [33]. ◁

In this first subsection, we present (without details) the definition of simulation as presented in the original paper. This definition is based on a notion of environment equivalence, which we present first[5]. Intuitively, two environments are equivalent if they allow and forbid the same transitions.

This notion is mandatory to take into account the fact that two different graphs change the environment of the rule, yet they can have the same behaviour (the same observed labels).

Definition 42 (Environment equivalence)**.** Environment equivalence, noted $\Gamma \Rightharpoonup \Gamma'$, is defined as:

$$\Gamma \Rightharpoonup \Gamma' \triangleq \forall \Upsilon \in \mathbb{E} \cdot \Gamma \cup \Upsilon \in \mathbb{E} \wedge \Gamma' \cup \Upsilon \in \mathbb{E} \Rightarrow \forall t \cdot \mathtt{Auth}(\Gamma \cup \Upsilon, t) = \mathtt{Auth}(\Gamma' \cup \Upsilon, t)$$

We extend that definition to the location graphs themself.

Definition 43.

$$G \Rightharpoonup H \triangleq \mathtt{supp}(G) \cdot \Sigma(G) \Rightharpoonup \mathtt{supp}(H) \cdot \Sigma(H)$$

With this definition of environment equivalent, we can now define similar graphs. Finally, a relation $\mathcal{R}$ over graphs is a (strong) simulation if and only if:

Definition 44 (Location graph simulation)**.** A relation $\mathcal{R} \subseteq \mathbb{G} \times \mathbb{G}$ is a strong simulation if and only if, for all $\langle G, F \rangle \in \mathcal{R}$, for all graph G' and for all label Λ, the following properties hold:

(i) $G \Rightharpoonup F$; and
(ii) $G.\mathtt{punbound} = F.\mathtt{punbound}$; and
(iii) $G.\mathtt{runbound} = F.\mathtt{runbound}$; and
(iv) Transitions of G are matched by F, i.e.

$$\begin{aligned} \forall \Gamma \in \mathbb{E} \cdot (\Gamma \cup \mathtt{supp}(G) \cdot \Sigma(G) \in \mathbb{E}) \wedge (\Gamma \cup \mathtt{supp}(F) \cdot \Sigma(F) \in \mathbb{E}) \\ \Rightarrow \Gamma \cup \mathtt{supp}(G) \cdot \Sigma(G) \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \\ \Rightarrow \exists F' \cdot \Gamma \cup \mathtt{supp}(F) \cdot \Sigma(F) \vdash_{\mathcal{T}_u} F \xrightarrow{\Lambda} F' \wedge \langle G', F' \rangle \in \mathcal{R} \end{aligned}$$

This definition has an important limitation: it relates graphs from the same model (i.e. with the same sets for processes, messages, etc. .). Our goal is to formalise a notion of encapsulation for location graphs. Said otherwise, ultimately, we want to have a *simulation* relation between a plain graph and its

[4] We tried an approach in which we built a common instance for both flat and encapsulated graphs. This approach is presented in Appendix B. However this approach was not successful, and we therefore stuck to an heterogenenous approach.

[5] Note that this notion of environment equivalence is not satisfying either. We use it only in this section. We will later (in Definition 45) define a more suitable environment equivalence. Therefore, it should not be used elsewhere.

encapsulated counterpart. We can already expect that processes and sorts will not be the same. Similarly, we will probably need some additional messages.

This implies that we need a more general notion of simulation, which accommodate different models. We qualify such simulation *heterogeneous*.

3.2.2 Heterogeneous simulations

In this section, we improve the definition of simulation (and the dependent definition) to fit relations $\mathcal{R}$ over $\mathbb{G}_1 \times \mathbb{G}_2$, for $\mathbb{G}_1 = \texttt{lgraph}(\mathbb{P}_1, \mathbb{S}_1, \mathbb{R}_1)$ and $\mathbb{G}_2 = \texttt{lgraph}(\mathbb{P}_2, \mathbb{S}_2, \mathbb{R}_2)$.

In this section, we are given $\mathbb{P}_1$ and $\mathbb{P}_2$ two sets of processes; $\mathbb{S}_1$ and $\mathbb{S}_2$ two sets of sorts; $\mathbb{R}_1$ and $\mathbb{R}_2$ two sets of roles; $\mathbb{C}_1$ and $\mathbb{C}_2$ two sets of channels; and $\mathbb{V}_1$ and $\mathbb{V}_2$ two sets of values.

Let $\mathbb{G}_1 = \texttt{lgraph}(\mathbb{P}_1, \mathbb{S}_1, \mathbb{R}_1)$; $\mathbb{A}_1 = \texttt{sLabel}(\mathbb{R}_1, \mathbb{C}_1, \mathbb{V}_1)$; $\mathcal{T}_1 \in \texttt{trans}(\mathbb{G}_1, \mathbb{A}_1)$; and $\texttt{Auth}_1 \in \texttt{sAuth}(\mathbb{G}_1, \mathbb{A}_1)$.

Let $\mathbb{G}_2 = \texttt{lgraph}(\mathbb{P}_2, \mathbb{S}_2, \mathbb{R}_2)$; $\mathbb{A}_2 = \texttt{sLabel}(\mathbb{R}_2, \mathbb{C}_2, \mathbb{V}_2)$; $\mathcal{T}_2 \in \texttt{trans}(\mathbb{G}_2, \mathbb{A}_2)$; and $\texttt{Auth}_2 \in \texttt{sAuth}(\mathbb{G}_2, \mathbb{A}_2)$.

Equivalence relations. We are given relations between elements of the first model and elements of the second model. These relations are parameters of the simulation.

We are given a relation $\eqsim_E$ over environments and a relation $\eqsim_\lambda$ over labels.

Definition 45 (Environment equivalence)**.** An *environment equivalence relation* is a relation $\eqsim_E \subseteq (\mathbb{A}_1 \times \mathbb{G}^{\mathsf{s}}{}_1) \times (\mathbb{A}_2 \times \mathbb{G}^{\mathsf{s}}{}_2)$ such that $\forall (\Gamma_1, \Gamma_2), (\Gamma_1', \Gamma_2') \in \eqsim_E \cdot \Gamma_1 \cup \Gamma_1' \eqsim_E \Gamma_2 \cup \Gamma_2'$

Remark. We draw the reader's attention to the fact that our definition of $\eqsim_E$ is not exactly the same than the one used for the original simulation. ◁

Remark. Notice that, contrary to the original definition of simulation, in our definition the environment equivalence relation is a parameter of the simulation. This let the user have a finer touch on how to relate two location graphs. ◁

Simulation, bisimulation.

Definition 46 (Heterogenous simulation)**.** Given an environment equivalence relation $\eqsim_E$ and a relation $\eqsim_\lambda \subseteq \mathbb{A}_1 \times \mathbb{A}_2$,
A relation $\mathcal{R} \subseteq \mathbb{G}_1 \times \mathbb{G}_2$ is a $\langle \eqsim_E, \eqsim_\lambda \rangle$-*strong simulation* if and only if, for all $\langle G_1, G_2 \rangle \in \mathcal{R}$, for all environment $\Delta_1 \cdot G_1^s$, for all G_1' such that $\Delta_1 \cdot G_1^s \cup \Sigma(G_1) \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} G_1'$,

(i) $\exists (\Delta_2 \cdot G_2^s) \cdot \Delta_1 \cdot G_1^s \eqsim_E \Delta_2 \cdot G_2^s \wedge \texttt{supp}(G_2) \subseteq \Delta_2 \wedge \Sigma(G_2) \subseteq G_2^s$

(ii) $\exists \Lambda_2 \cdot \Lambda_1 \eqsim_\lambda \Lambda_2$

(iii) $\forall (\Delta_2 \cdot G_2^s) \cdot (\Delta_1 \cdot G_1^s) \eqsim_E (\Delta_2 \cdot G_2^s) \wedge (\texttt{supp}(G_2) \subseteq \Delta_2) \wedge (\Sigma(G_2) \subseteq G_2^s) \Rightarrow \exists G_2' \cdot \Delta_2 \cdot (G_2^s \cup \Sigma(G_2)) \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G_2'$

(iv) $\langle G_1', G_2' \rangle \in \mathcal{R}$

(v) $\texttt{supp}(G_1') \cdot \Sigma(G_1') \eqsim_E \texttt{supp}(G_2') \cdot \Sigma(G_2')$

This definition of simulation is a bit more involved than usual. This is due to the fact that we want to compare location graphs of different models. For that, we require a notion of *equivalent* environments (given by the $\eqsim_E$ relation), but this relation is not necessary one-to-one.
Actually, we are fine if an environment of the first model does not have an equivalent environment in the second model, as long as we do not use that environment in our reductions. Therefore, we only need that $\exists (\Delta_2 \cdot G_2^s) \cdot \Delta_1 \cdot G_1^s \eqsim \Delta_2 \cdot G_2^s$ in the condition, and not as a general requirement for $\eqsim_E$.
The second question that may arise could be: why do we have the existential constraint in (i), and then a universal quantifier in (iii)? Why could not we write directly a constraint like:

$$\exists (\Delta_2 \cdot G_2^s) \cdot (\Delta_1 \cdot G_1^s) \eqsim_E (\Delta_2 \cdot G_2^s) \wedge \Delta_2 \cdot (G_2^s \cup \Sigma(G_2)) \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G_2'$$

The answer is not trivial, and stands in two steps: (i) the goal of simulations is to relate location graphs that behave alike, and therefore, the intuition we want to catch is that if two graphs are similar, and if the first takes a transition, the second must be able to take a similar transition, and both new location graphs should be similar; and (ii) one can not freely choose the skeleton location graphs involved in the environment, this skeleton graph should contain, by definition of the reduction rules, the skeleton of the overall location graph.
The consequence of the first point is that, if we have $\Delta_1 \cdot G_1^s \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} G_1'$ and $\Delta_1' \cdot G_1^{s\prime} \vdash_{\mathcal{T}_1}$

$G'_1 \xrightarrow{\Lambda_1} G''_1$, we want to show the existence of the two following transitions: $\Delta_2 \cdot G_2^s \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G'_2$ and $\Delta'_2 \cdot G_2^{s\prime} \vdash_{\mathcal{T}_2} G'_2 \xrightarrow{\Lambda_2} G''_2$. The point is that, at the point where we have G'_1 and G'_2, we have to remember that they are potentially part of a bigger graph, say $G'_1 \parallel G_1^r$ and $G'_2 \parallel G_2^r$, and by definition of the reduction rules, $\Sigma(G'_1 \parallel G_1^r)$ (resp. $\Sigma(G'_2 \parallel G_2^r)$) are in $G_1^{s\prime}$ (resp. $G_2^{s\prime}$): we can not freely choose those two environments (the second point); and therefore, we have to find a way to enforce that we can always find such environment.
Intuitively, we want to state that "after each transition taken, the new environments we arrive in are in $=_E$".
Without such convoluted definition, it is hard to compare two different location graph instances. For instance, as stated above, in Appendix B, we tried to compare two location graph instances by constructing a common third instance, which could implement both instances, and then analysing the relationship between the two instances with the regular simulation on location graphs. Such approach was not successful since, for instance, the standard simulation definition compares location graphs in the same environment, which is not our case; hence our approach which introduces a comparison up-to a chosen relation on environments.

Definition 47 (Strong bisimulation)**.** A relation $\mathcal{R}$ is a *strong* $\langle =_E, =_\lambda \rangle$*-bisimulation* if and only if $\mathcal{R}$ is a strong $\langle =_E, =_\lambda \rangle$-simulation and $\mathcal{R}^{-1}$ is a strong $\langle =_E^{-1}, =_\lambda^{-1} \rangle$-simulation.

One can clearly see that, with a careful definition of $=_E$, and by taking the equality for $=_\lambda$ our new definition of heterogeneous simulation is conservative with the original definition.

Our simulation is not to be confused with the notion of environmental bisimulation developed by Sangiorgi *et al.* in [52].
In their paper, they present a new kind of simulation, called environmental simulation. The idea is to take an additional relation over values, which are then considered equivalent (i.e. two terms do not necessarily reduce to the same value, but to equivalent values). In addition, they put some constraint on equivalent value: informally, the set of equivalent values should be closed under substitution (if $\lambda x.P$ and $\lambda x.Q$ are equivalent, and M and N too, then $P\{M/x\}$ and $Q\{N/x\}$ should be equivalent).

3.2.3 Partial bisimulation

The notion of *partial bisimulation* was first[6] introduced by Rutten in [49]. This variant of bisimulation lays inbetween a simulation and a bisimulation: for each $\langle a, b \rangle$ in the relation, a simulates b but, as opposed to the usual bisimulation, b simulates a only for some labels.

Here, we define partial bisimulation for graphs of different models (external approach). The same could be done for internal simulations.

Definition 48 (Partial bisimulation)**.** A $\mathcal{R} \subseteq \mathbb{G}_1 \times \mathbb{G}_2$ is a *partial bisimulation* with respect to $\mathbb{U} \subset \mathbb{A}_2$ if and only if, for any $\langle G_1, G_2 \rangle \in \mathcal{R}$:

(i) $\forall G'_1, \Gamma_1, \Lambda_1$ if $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} G'_1$ then $\exists G'_2, \Gamma_2, \Lambda_2$ such that:
  (i) $\exists (\Delta_2 \cdot G_2^s) \cdot \Delta_1 \cdot G_1^s =_E \Delta_2 \cdot G_2^s \wedge \mathtt{supp}(G_2) \subseteq \Delta_2 \wedge \Sigma(G_2) \subseteq G_2^s$
  (ii) $\exists \Lambda_2 \cdot \Lambda_1 =_\lambda \Lambda_2$
  (iii) $\forall (\Delta_2 \cdot G_2^s) \cdot (\Delta_1 \cdot G_1^s) =_E (\Delta_2 \cdot G_2^s) \wedge (\mathtt{supp}(G_2) \subseteq \Delta_2) \wedge (\Sigma(G_2) \subseteq G_2^s) \Rightarrow \exists G'_2 \cdot \Delta_2 \cdot (G_2^s \cup \Sigma(G_2)) \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G'_2$
  (iv) $\langle G'_1, G'_2 \rangle \in \mathcal{R}$
  (v) $\mathtt{supp}(G'_1) \cdot \Sigma(G'_1) =_E \mathtt{supp}(G'_2) \cdot \Sigma(G'_2)$

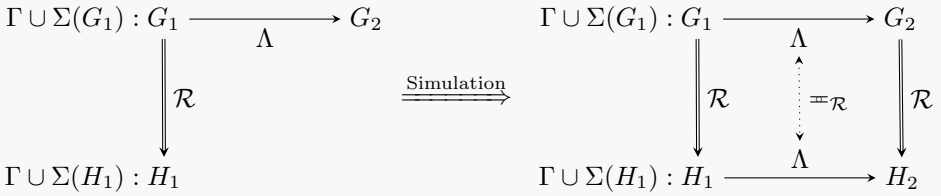
(ii) $\forall G'_2, \Gamma_2, \Lambda_2$ if $\Lambda_2 \in \mathbb{U}$ and $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G'_2$, then $\exists G'_1, \Gamma_1, \Lambda_1$ such that:
  (i) $\exists (\Delta_1 \cdot G_1^s) \cdot \Delta_1 \cdot G_1^s =_E \Delta_2 \cdot G_2^s \wedge \mathtt{supp}(G_1) \subseteq \Delta_1 \wedge \Sigma(G_1) \subseteq G_1^s$
  (ii) $\exists \Lambda_1 \cdot \Lambda_1 =_\lambda \Lambda_2$
  (iii) $\forall (\Delta_1 \cdot G_1^s) \cdot (\Delta_1 \cdot G_1^s) =_E (\Delta_2 \cdot G_2^s) \wedge (\mathtt{supp}(G_1) \subseteq \Delta_1) \wedge (\Sigma(G_1) \subseteq G_1^s) \Rightarrow \exists G'_1 \cdot \Delta_1 \cdot$

[6] To the best of my knowledge.

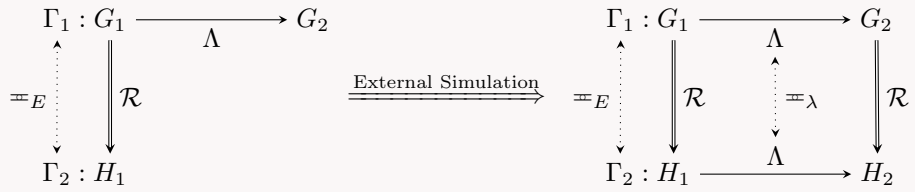
$$(G_1^s \cup \Sigma(G_1)) \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} G_1'$$
(iv) $\langle G_1', G_2' \rangle \in \mathcal{R}$
(v) $\mathtt{supp}(G_1') \cdot \Sigma(G_1') =_E \mathtt{supp}(G_2') \cdot \Sigma(G_2')$

Summary – Location Graph Simulations

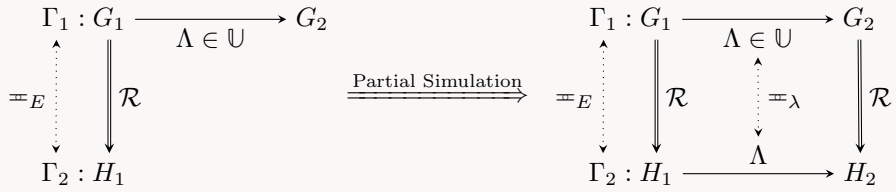
In this section, we introduced different notions of bisimulation. To better illustrate the differences between those notions, this summary shows the diagrams of each of the bisimulation.
A location graph H_1 is similar to G_1 if, for any environment Γ, they can reduce to similar graphs G_2 and H_2, with the same label Λ: we can close the square.



We saw that the regular simulation is to restrictive. The *external simulation*, on the other hand, allows us to close the square when the environement and the label are related (by $=_E$ and $=_\Lambda$).



Finally, we want to be able to ignore some messages (typically administrative messages). Therefore, partial simulation requires that only some messages (those in $\mathbb{U}$) are concerned by the simulation.



3.3 Nested Location Graphs

Our main goal is to study encapsulation policies, and to show that our policies behave correctly. The difficulty, of course, is that there is not a unique notion of *behaving correctly*, as we have already shown. Therefore, we take an other approach, which consists in exhibiting, for each policy, an equivalent graph where encapsulation aggregates are grouped in a single location: said otherwise, we switch from a location graph which represent a graph of objects to a location graph which represent a graph of aggregates. Such procedure allows the analysis of communications at the aggregate level.

Our approach is to build location graphs whose processes are location graphs themself (i.e. nested location graphs). With such a definition, we can simply split a location graph into subgraphs, one per aggregate, and to nest those subgraphs into individual locations.

In the previous section, we introduced a notion of partial bisimulation. The goal of this section is two fold: (i) presenting a generic mechanism for nesting location graphs; and (ii) proving that this mechanism is faithful, with respect to the (partial) bisimulation we defined previously.

The nesting mechanism we introduce has to be generic to accommodate any encapsulation policy. We therefore make no assumption about the underlying graph. The notion of encapsulation being quite informal, so is the nesting mechanism: it does not provide any isolation guarantee by itself: instead, it gives an new view of a system, where some details are abstracted away.

For the sake of clarity, we call *1st order graph* the original graph, and *2nd order graph* a graph that

nests the original graph. In the last subsection of this section, we informally draw some perspectives to extend our work to allow an arbitrary number of nesting levels.

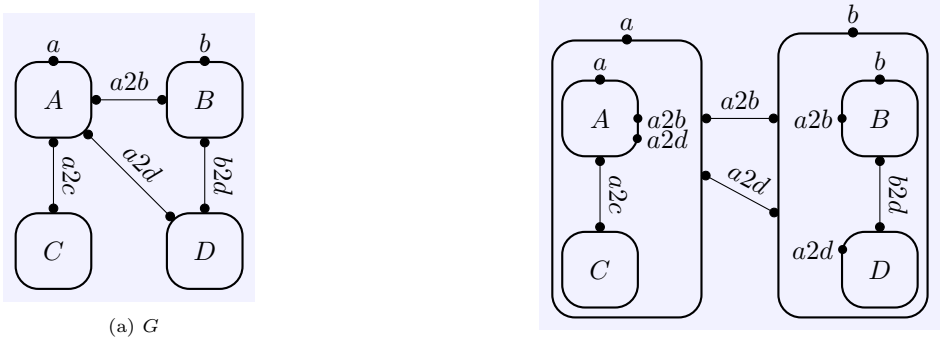
Informal presentation. To understand the intuition of our method of nesting, let us forget, for one second, everything we learnt so far about location graphs, and let us start again our initial presentation of location graphs, from a graphical point of view: we have circles, which we call *locations*, and dots and lines on those circles, which we call unbound and bound *role*. Nesting is, essentially, grouping together some of those circles. Graphically, we can circle groups of circles, report to the outer circle all dots that are on inner circles, and finally blur the internal details of each outer circle. Lines that do not cross outer circles are entirely blurred, and lines that do cross outer circles are (partially) visible.

We would result with a new set of circles with dots and lines, i.e. a location graph. Also, no matter how the internal parts of each outer circle evolve, we can always draw those outer circles. *The essence of nesting, such as presented in this section, is a matter of drawing circles.*

Of course, we do not want to draw those circles at random: we want an instruction that tells us how to split the graph: we call this instruction the *partition function*.

The two important results of this section can also be stated informally. The first one is that the way we draw the circles does not really matter: from the same original graph you may draw some circles, I may draw them differently, but the underlying graph still evolves in the same way. We do not blur the same area, we may not see the same lines between outer circles, yet, the underlying graph is the same. The second important result is that, fundamentally, we are not doing much: with or without outer circles, the behaviour is essentially the same, we are just ignoring some parts of it.

In Figure 3.2, a 1st order location graph is shown, as well as a possible corresponding 2nd order graph.



(a) G

(b) G nested

Figure 3.2: A 1st order graph G (Subfigure 3.2a) and a 2nd order graph that embeds G (Subfigure 3.2b). Notice how bound roles of a subgraph (e.g. $a2c$) do not appear in the roles of the outer location. Unbound roles, however, are also unbound in the 2nd order graph, that is role bindings between outer locations are preserved (e.g. $a2b$).

This section contains three main parts: (i) in Section 3.3.1, we describe how to nest a graph into a location. In that section, we are not interested in the dynamic aspects of location graphs, we simply explain the structure of 2nd order graphs. Intuitively, our 2nd order locations encompass a subgraph of the 1st order graph. Therefore, we have two things to do: building a location which contains an arbitrary graph; and splitting a graph into multiple pieces.; (ii) in Section 3.3.2, we now study the dynamic aspects of 2nd order graphs, i.e. we describe their semantics. According to the location graph framework, we proceed in two steps: we first describe the set of unconstrained (2nd order) location transitions; and then, we describe the authorisation function for the 2nd order graph. We conclude this section with one of most important result[7] of this work: Lemma 23, which states that our semantics allows us to change the way the 1st order graph is split and nested at will.; and (iii) the first two sections give us a method to build a 2nd order graph and its semantics, but they do not say anything about the relation between the 1st and 2nd order graphs. In Section 3.3.3, we actually show that the two previous section make sense, in the sense that the 1st and 2nd order graphs are in a partial bisimulation relation.

The conclusion of all this work is that the location graph framework is general enough to describe nesting. The three sections laid out above can easily be extended to a finite number of layers, provided

[7] Actually, I personally think this result is even more important than the bisimulation result that follows; for two reasons: (i) the proof bisimulation result heavily relies on that lemma; and (ii) the proof of this lemma is actually much nicer and more readable than that of the bisimulation result.

that we do not mix (sub)graphs which are nested at different levels. In the last section (Section 3.3.4), we discuss informally the possibility of actually mixing graphs nested at different levels.

3.3.1 Nesting functions

In the introduction of this section, we informally presented our notion of nesting, as drawing circles around some locations of a given graph. We now focus on the formal aspects of this procedure.

In a first step, we explain how a (sub)graph can be nested in a single location. We also show that our method allows to retrieve the original graph from its nested counterpart.

In a second step, we explore *graph partitions*, which we use to split a location graph into pieces. This part is similar to defining partitions on multisets. Yet, even if the theory exists for multisets, it does not for location graph. The most important result of this paragraph is that partitions form a lattice. This is, of course, not surprising, but it is an important point, latter, for the proof of Lemma 23. Finally, we merge these two steps, to present the overall graph nesting function, which basically partitions a graph and nests the elements of the partition.

Location nesting. Let begin with the nesting of a graph G. We construct a (single) location, whose process is G and whose sort is the skeleton of G. The roles of the location are the unbound roles of G, plus some administrative roles.

The intuition behind the reason why we need to keep the skeleton of G in the sort is that the semantics of the 2^{nd} order graph, which we will detail below, are based on the semantics of the 1^{st} order graph, which requires the skeleton of 1^{st} order graph (e.g. , to evaluate the authorisation function).
By putting the skeleton of nested graph in the sort, it also appears in the skeleton of the 2^{nd} order graph (the skeleton only drops the processes), which allows us to recover the 1^{st} order skeleton graph, and then evaluate its reductions, to find the reduction of the 2^{nd} order graph.

Definition 49 (Location nesting function)**.** Given a location graph G, we define the function

$$n(G) \triangleq \{[G : \Sigma(G) \lhd p \bullet r] | p = G.\texttt{punbound} \cup P \wedge r = G.\texttt{runbound} \cup R \wedge P, R \subset \mathbb{R}_2 \wedge P \cap R = \emptyset\}$$

Remark (Process and sort equality). Processes are now 1^{st} order graphs and sorts are 1^{st} order skeleton graphs. Hence, *stricto sensu*, if $G_1 \equiv G_2$ with $G_1 \neq G_2$, then $n(G_1) \neq n(G_2)$. To avoid that, we take the location graph structural congruence ($\equiv$) as processus equality and skeleton graph structural congruence ($\equiv$) as sort equality.

Said otherwise, stricto sensu, the process (resp. the sort) of a second order location is the equivalence class of a first order graph (resp. first order skeleton).

An other possible approach, which we do not take, could be to define an equivalence relation over 2^{nd} order locations $\mathcal{L} \triangleq \{\langle L_1, L_2 \rangle | L_1.\texttt{proc} \equiv L_2.\texttt{proc}\}$, and to define $n([\cdot])$ to be the equivalence class of the nested graph. This would however be less practical, morally, as we would have to think of the semantics as the semantics of an equivalence class of location graphs instead of the semantics of a location graph.

It turns out that, in the following of this thesis, we are never bothered by these details, and we just mention them here to be pedantic. Of course, a (computer assisted) verification of this work could not make that (abusive) simplification. ◁

Remark. Naturally, all thusly formed location are well formed: $\forall L \in n(G), \texttt{WF}(L)$ ◁

Remark (Administrative roles of nested graphs). In the definition, P and R are additional administrative (provided and required) roles. Here, we do not specify exactly the number and the names of those roles. In the following of this thesis, we assume we always have enough of those roles and we use them at will. Therefore, when we note $n(G)$, we implicitly mean an actual element of $n(G)$, instead of the set itself. We assume this element always have enough administrative roles to do what we intend to do (the worst case being one administrative role toward and from any other location of the graph). ◁

The correspondence between the elements of G and $G_2 \in n(G)$ is shown in Table 3.1. We assume $\mathbb{R}_2$ is disjoint of $\mathbb{R}$ (it contains fresh role names which are used for *administrative communications*, see below).

An interesting property is that, from any element of $n(G)$, we can recover the original G. We note the function that does that $n^{-1}([\cdot])$, defined as:

	G	$G_2 = n(G)$
Processes	$\mathbb{P}$	$\mathbb{G}$
Sorts	$\mathbb{S}$	$\mathbb{G}_s$
Roles	$\mathbb{R}$	$\mathbb{R} \cup \mathbb{R}_2$
Channels	$\mathbb{C}$	$\mathbb{C}$
Values	$\mathbb{V}$	$\mathbb{V} \cup \{n(L)\}$ for $L \in \mathbb{L}$
Locations	$\mathbb{L}$	$\mathbb{L}_2$
Location graphs	$\mathbb{G}$	$\mathbb{G}_2$

Table 3.1: Relation between components of a location graph before and after nesting.

Definition 50 (Inverse of location nesting)**.**

$$n^{-1}(L) \triangleq L.\texttt{proc}$$

Lemma 11 (Soundness of the inverse of location nesting)**.**

$$\forall G \in \mathbb{G} \cdot G = n^{-1}(n(G))$$

Proof. The proof follows directly from Definitions 49 and 50:

$$n^{-1}(n(G)) = [G : \Sigma(G) \lhd p \bullet r].\texttt{proc} = G$$

□

Graph partitions. As stated in the introduction, the second point to nest graphs is to find a way to split graphs into subgraphs. To do so, we define *graph partitions* and *graph partitioning functions*, inspired from (multi)set partitions. In the second part of this paragraph, we define a partial order relation on partitions, and we show that the partitions of a graph form a lattice.

First, a partition of a graph is a multiset of subgraphs: it is a multiset of location graphs, such that when we compose all elements of the partition, we fall back on (an equivalent of) the initial graph.

Definition 51 (Graph partition)**.** Let $G \in \mathbb{G}$ be a location graph. Let $\mathbb{G}_i$ be a multiset of elements (location graphs) of $\mathbb{G}$. $\mathbb{G}_i$ is a partition of G if and only if
(i) $G \equiv \prod_{G_i \in \mathbb{G}_i} G_i$; and
(ii) $\forall G_i \in \mathbb{G}_i, G_i \not\equiv \emptyset$.
The set of partitions of a graph G is noted $\mathbb{P}_G$.

Remark (Partition equivalence). Strico sensus, given a graph $G = G_1 \parallel G_2$ and a $G_1' \equiv G_1$, the partitions $\{G_1, G_2\}$ and $\{G_1', G_2\}$ are distinct.

To overcome this, we implicitly distinguish partitions up to the equivalence of their elements. ◁

In general, we want to describe a way to have a partition from *any* graph. Therefore, we define *graph partitioning functions* which are functions that take a location graph and return a partition of that graph.

Definition 52 (Graph partitioning function)**.** A function $p : \mathbb{G} \to \mathcal{P}(\mathbb{G})$ is a partitioning function over $\mathbb{G}$ if and only if:

$$\forall G \in \mathbb{G}, p(G) \in \mathbb{P}_G$$

The set of graph partitioning functions over $\mathbb{G}$ is noted $\mathbb{P}_{\mathbb{G}}$.

Remark. In the following, we only consider partitioning functions that are invariant under structural equivalence, that is, we only consider partitioning functions p such that, for all graphs G_1 and G_2, $G_1 \equiv G_2 \Rightarrow p(G_1) = p(G_2)$.

Notice that, in general, this is not necessary the case. ◁

Now that we have definitions for partitions and partitioning functions, we anticipate the future a bit[8] and we present a notion of *finer* and *coarser* partitions.

Intuitively, we say that a partition p_1 is finer than p_2 (or that p_2 is coarser) if it is possible to recover p_2 by merging some elements of p_1. Said otherwise, elements of p_2 are aggregates of elements of p_1.

Defining what is a *finer* or *coarser* partition of a location graph with respect to an other partition is not as easy as it may seem at first glance. Intuitively, we would define this relation as:

> **Example 4** (Bad definition of *finer*)**.** Given a location graph G and $\mathbb{G}_1$ and $\mathbb{G}_2$ two partitions of G, $\mathbb{G}_1$ is *finer* than $\mathbb{G}_2$ (noted $\mathbb{G}_1 \succ \mathbb{G}_2$) if and only if:
>
> $$\forall G_i \in \mathbb{G}_1 \cdot \exists G'_i \in \mathbb{G}_2 \cdot G_i \subseteq G'_i$$

However, this definition is not sufficient since it is possible to have duplicate locations. An example which illustrate the problem is the following: consider the graph $G = L_1 \parallel L_2 \parallel L_1 \parallel L_2$ and the partitions $\mathbb{G}_1 = \{L_1 \parallel L_2; L_1 \parallel L_2\}$ and $\mathbb{G}_2 = \{L_1 \parallel L_2; L_1; L_2\}$.
Here, we would intuitively say that $\mathbb{G}_2$ is finer than $\mathbb{G}_1$ (which is true) but we would not say that $\mathbb{G}_1$ is finer than $\mathbb{G}_2$. Unfortunately, this does not hold here: both $L_1 \parallel L_2$ in $\mathbb{G}_1$ are subsets of *the unique* $L_1 \parallel L_2$. The individual L_1 and L_2 of $\mathbb{G}_2$ are not *used* in the mapping.
Therefore, we say that $\mathbb{G}_1$ is finer than $\mathbb{G}_2$ if $\mathbb{G}_1$ is obtained by partitioning the subgraphs in $\mathbb{G}_2$. However, the intuitive formalisation of this is to say that

$$\mathbb{G}_1 = \bigcup_{G_i \in \mathbb{G}_2} p(G_i)$$

for some partition function p. This is not satisfying because two equivalent subgraphs in $\mathbb{G}_2$ should not necessarily be partitioned similarly. Therefore, in the following definition, we use a different partitioning function p_i for each G_i in $\mathbb{G}_2$.

Definition 53 (Finer graph partition)**.** Given a location graph G and two partitions $\mathbb{G}_1$ and $\mathbb{G}_2$ of G, $\mathbb{G}_1$ is *finer* than $\mathbb{G}_2$ (noted $\mathbb{G}_1 \succ \mathbb{G}_2$) if and only if[a] there exists a set of partition functions p_i such that:

$$\mathbb{G}_1 = \bigcup_{G_i \in \mathbb{G}_2} p_i(G_i)$$

If $\mathbb{G}_1 \succ \mathbb{G}_2$, then $\mathbb{G}_2$ is a *coarser graph partition* than $\mathbb{G}_1$ (noted $\mathbb{G}_2 \prec \mathbb{G}_1$).

[a] Notice the *union* used is the union of multisets.

This definition implies the intuitive view of partition we have:

Lemma 12.

$$\mathbb{G}_1 \succ \mathbb{G}_2 \Rightarrow \forall G_j \in \mathbb{G}_1 \cdot \exists G_i \in \mathbb{G}_2 \cdot G_j \subseteq G_i$$

Proof. From Definition 53, $\mathbb{G}_1 = \bigcup_{G_i \in \mathbb{G}_2} p_i(G_i)$ for some partition functions p_i.
For any element G_j of $\mathbb{G}_1$, $\exists G_i \in \mathbb{G}_2 \cdot G_j \in p_i(G_i)$. By definition of partition function (Definition 52) and partition (Definition 51), $G_i \equiv \prod_{G_k \in p_i(G_i)} G_k$, i.e. $G_i \equiv G_j \parallel G_k$ for $G_k \equiv \prod_{G_k \in p_i(G_i) \setminus G_j} G_k$.
Therefore, from Lemma 7, $G_j \subseteq G_i$. □

An other important property about $p_1 \succ p_2$ is that if p_1 and p_2 share a common element, then we can remove that element on both partitions, and they remain related.

[8] Thanks to the very nature of manuscripts, the puzzled reader can actually *see* the future, by having a look at the proofs of Lemmas 21 and 22, which are the two sides of Lemma 23.

Lemma 13. *Given a graph G and two partitions $\mathbb{G}_1$ and $\mathbb{G}_2$ of G such that $\mathbb{G}_1 \succ \mathbb{G}_2$.*

$$\forall G_0 \cdot G_0 \in \mathbb{G}_1 \wedge G_0 \in \mathbb{G}_2 \Rightarrow \mathbb{G}_1 \setminus G_0 \succ \mathbb{G}_2 \setminus G_0$$

Proof. From Definition 53, $\mathbb{G}_1 = \bigcup_{G_i \in \mathbb{G}_2} p_i(G_i)$ for some partition functions p_i.
Since $G_0 \in \mathbb{G}_1$, there exists a $G_j \in \mathbb{G}_2$ such that $G_0 \in p_j(G_j)$. Also, since $G_0 \in \mathbb{G}_2$, there exists a $G_k \equiv G_0 \in \mathbb{G}_2$.
Therefore,

$$\mathbb{G}_1 = \bigcup_{G_i \in \mathbb{G}_2} p_i(G_i) = \bigcup_{\substack{G_i \in \mathbb{G}_2 \\ i \neq j,k}} p_i(G_i) \cup (p_j(G_j) \setminus G_0) \cup p_k(G_k) \cup G_0$$

We have $p_j(G_j) = G_0 \cup (p_j(G_j) \setminus G_0)$, since p_j is a partition function, $G_j \equiv \prod_{G \in p_j(G_j)} G$. In addition, since p_k is a partition function, $G_0 \equiv \prod_{G \in p_k(G_0)} G$.
Therefore, $G_j \equiv \prod_{G \in p_j(G_j) \setminus G_0} G \cup \prod_{G \in p_k(G_0)} G$. Thus $p_j(G_j) \setminus G_0 \cup p_k(G_0)$ is a partition of G_j.
Let $p_{jk}(G)$ be a function that returns p_j if $G \neq G_j$ and $p_j(G_j) \setminus G_0 \cup p_k(G_0)$ if $G = G_j$.
Let $p'_1, \ldots$ be a family of partition function such that $p'_i = p_i$ if $i \neq j$ and $p'_j = p_{jk}$.
Finally[a],

$$\mathbb{G}_1 \setminus G_0 = \bigcup_{\substack{G_i \in \mathbb{G}_2 \setminus G_0 \\ i \neq j}} p_i(G_i) \cup (p_j(G_j) \setminus G_0 \cup p_k(G_k)) = \bigcup_{G_i \in \mathbb{G}_2 \setminus G_0} p'_i(G_i)$$

Therefore $\mathbb{G}_1 \setminus G_0 \succ \mathbb{G}_2 \setminus G_0$. □

[a] Notice that, since G_0 is removed, the index k in the union is removed also.

We can now show that $\succ$ is a partial order. The only non-trivial part is antisymmetry, as it is possible to have the same subgraph multiple times in the partition (for instance if the original graph has multiple copies of the same location).

Lemma 14. *The finer relation is a partial order.*

Proof. Reflexivity: Trivial.

Antisymmetry: We have $\mathbb{G}_1$ and $\mathbb{G}_2$ with $\mathbb{G}_1 \succ \mathbb{G}_2$ and $\mathbb{G}_2 \succ \mathbb{G}_1$. We show that $\forall G_i \in \mathbb{G}_1, \exists G_i \in \mathbb{G}_2$, the other direction holds by symmetry. Since a location graph contains a finite number of location and since location graph partitions do not contain empty graphs elements, $\mathbb{G}_1$ and $\mathbb{G}_2$ have a finite number of elements. By induction on the number of elements of $\mathbb{G}_1$ (induction hypothesis: $\forall \mathbb{G}_1, \mathbb{G}_2 \cdot \mathbb{G}_1 \succ \mathbb{G}_2 \wedge \mathbb{G}_2 \succ \mathbb{G}_1 \Rightarrow \mathbb{G}_1 = \mathbb{G}_2$):
Base case ($|\mathbb{G}_1| = 0$): Vacuously holds.

Inductive case ($|\mathbb{G}_1| = n + 1$): Let $G_0 \in \mathbb{G}_1$ be such that $\forall G_i \in \mathbb{G}_1 \cdot \text{size}(G_0) \geq \text{size}(G_i)$. Since $\mathbb{G}_1 \succ \mathbb{G}_2$, there exists $G'_0 \in \mathbb{G}_2$ such that $G_0 \subseteq G'_0$. Since $\mathbb{G}_2 \succ \mathbb{G}_1$, there exists $G_1 \in \mathbb{G}_1$ such that $G'_0 \subseteq G_1$. By hypothesis, $\text{size}(G_0) \geq \text{size}(G_1)$, and $G_0 \subseteq G'_0$ and $G'_0 \subseteq G_1$. Hence, from Lemma 8, $\text{size}(G_0) = \text{size}(G'_0) = \text{size}(G_1)$. Hence, from Lemma 9 $G_0 \equiv G'_0 \equiv G_1$.
From Lemma 13, $\mathbb{G}_1 \setminus G_0 \succ \mathbb{G}_2 \setminus G_0$ and $\mathbb{G}_2 \setminus G_0 \succ \mathbb{G}_1 \setminus G_0$. Hence, from the induction hypothesis: $\mathbb{G}_1 \setminus G_0 = \mathbb{G}_2 \setminus G_0$. Therefore, $\mathbb{G}_1 = \mathbb{G}_2$.

Transitivity: Follows directly from the definition and the transitivity of location graph inclusion. □

The last step toward to prove that partitions of a graph equipped with $\cdot \succ \cdot$ form a lattice is to show that we have a biggest and a smallest element. This is trivial, the finest partition of a graph separates each individual locations, while the coarsest partitions the graph into a single element: the graph itself.

Lemma 15. *For any location graph G, there is a smallest and biggest partition $\mathbb{G}_{\min}$ and $\mathbb{G}_{\max}$ in $\mathbb{P}_G$, i.e. $\forall \mathbb{G} \in \mathbb{P}_G \cdot \mathbb{G}_{\max} \succ \mathbb{G} \wedge \mathbb{G} \succ \mathbb{G}_{\min}$.*

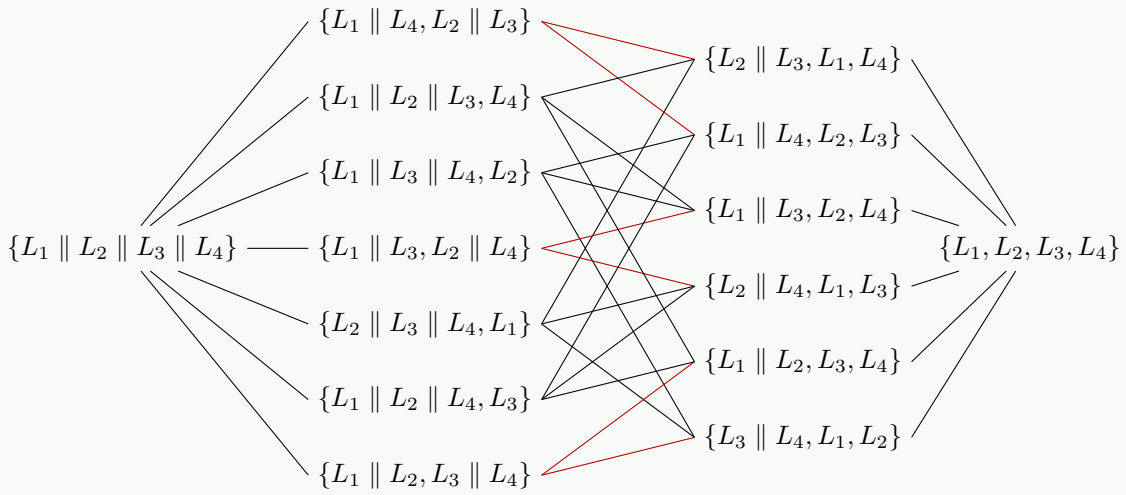
Proof. Smallest partition: Take the partition $\mathbb{G}_{\min} = \{G\}$.

BIGGEST PARTITION: Take $\mathbb{G}_{\max} = \text{locations}(G)$.

□

Corollary 1 (of Lemma 14 and 15). $(\mathbb{P}_G, \succ)$ form a lattice.

Example 5 (Lattice of partitions)**.** Given a location graph $G = L_1 \parallel L_2 \parallel L_3 \parallel L_4$, the following figure shows the lattice of the partitions of G.



Colours added to improve readability.

We previously defined finer and coarser graph partitions. One could extend this definition to partitioning functions, although we do not need that definition for the rest of the work.

Definition 54 (Finer/Coarser partitioning functions)**.** Given p_1 and p_2 two partitioning functions over $\mathbb{G}$, p_1 is *finer* than p_2 (noted[a] $p_1 \succ p_2$) if and only if:

$$\forall G \in \mathbb{G}, \forall G_i \in p_1(G), \exists G' \in p_2, G_i \subseteq G'$$

If $p_1 \succ p_2$, then p_2 is *coarser* than p_1 (noted $p_2 \prec p_1$).

[a] Intuitively, a subgraph of $p_2(G)$ can be split into multiple subgraphs in $p_1(G)$.

Graph nesting. Finally, we can define graph nesting. From the above material, our approach should be very intuitive: we partition a graph, and we nest the elements of the partition in individual locations. Notice that, as with $n()$, the function returns a set of possible nested graphs. The variants differ only by the chosen administrative roles and by equivalent graphs, we therefore ignore those differences.

Definition 55 (Graph nesting function)**.** A nesting function $N_{[\cdot]}([\cdot])$ is a function that takes a graph partitioning function p and a graph G and returns:

$$N_p(G) \triangleq \left\{ \prod_{G_i \in p(G)} G'_i | G'_i \in n(G_i) \right\}$$

Remark. Similarly to the remarks on $n([\cdot])$, by $N_p(G)$, we actually mean a particular element of $N_p(G)$, where we implicitly assume we have all the administrative roles we need. ◁

From any element $G_2 \in N_p(G)$, we can recover the original graph: each (2^{nd} order) location allows to recover the underlying element of the partition. By composing all elements of the partition recovered thusly, we can reconstruct the full 1^{st} order graph. We note the function that does that $N^{-1}_{[\cdot]}([\cdot])$

Definition 56 (Inverse of graph nesting)**.**

$$N_p^{-1}(G) \triangleq \prod_{L_i \in G} n^{-1}(L_i)$$

In that definition, all L_i are 2^{nd} order locations, i.e. locations that nest a subgraph of the original 1^{st} order graph.

Remark. $\forall p, p' \in \mathbb{P}_{\mathbb{G}}, N_p^{-1}(G) \equiv N_{p'}^{-1}(G)$. ◁

Lemma 16 (Soundness of $N_{[\cdot]}^{-1}([\cdot])$)**.**

$$\forall G \cdot \forall p \cdot N_p^{-1}(N_p(G)) \equiv G$$

Proof.

$$\begin{aligned} N_p^{-1}(N_p(G)) &\overset{\text{Def 56}}{=} \prod_{L_i \in N_p(G)} n^{-1}(L_i) \\ &\overset{\text{Def 55}}{=} \prod_{L_i \in \prod_{G_i \in p(G)} n(G_i)} n^{-1}(L_i) \\ &= \prod_{G_i \in p(G)} n^{-1}(n(G_i)) \\ &\overset{\text{Lem 11}}{=} \prod_{G_i \in p(G)} G_i \\ &\overset{\text{Def 52}}{\equiv} G \end{aligned}$$

□

We relate nested graph with the partitioning function used. We show that if we use two different partitioning function, we result in different 2^{nd} order graphs, and vice-versa.

Lemma 17. *For all G in $\mathbb{G}$, $N_p(G) = N_{p'}(G) \Leftrightarrow p(G) = p'(G)$.*

Proof. The direction $p(G) = p'(G) \Rightarrow N_p(G) = N_{p'}(G)$ is trivial.
To prove $N_p(G) = N_{p'}(G) \Rightarrow p(G) = p'(G)$, we prove $p(G) \neq p'(G) \Rightarrow N_p(G) \neq N_{p'}(G)$.
If $p(G) \neq p'(G)$, then $\exists G_i \in p(G) \cdot G_i \notin p'(G) \vee \exists G_i \in p'(G) \cdot G_i \notin p(G)$. By symmetry, we only consider the case $\exists G_i \in p(G) \cdot G_i \notin p'(G)$.
The location nesting function $n([\cdot])$ is injective, hence $n(G_i) \notin \{n(G_j) | G_j \in p'(G)\}$, which implies $n(G_i) \notin \prod_{G_j \in p'(G)} n(G_j)$.
Thus $\prod_{G_i \in p(G)} n(G_i) = N_p(G) \neq N_{p'}(G) = \prod_{G_j \in p'(G)} n(G_j)$. □

Corollary 2. If $G_1 \equiv G_2$, then, for any partition function p, $N_p(G_1) \equiv N_p(G_2)$.

Proof. This follows directly from the fact that we consider only partition functions that are invariant under structural equivalence and from Lemma 17. □

Finally, a last trivial lemma, which states that we do not lose unbound roles while nesting.

Lemma 18.

$$\forall G, p \cdot G.\texttt{unbound} \subseteq N_p(G).\texttt{unbound}$$

Proof. Trivial. □

As a conclusion, we now have a mechanism to build nested variants of any location graph. Aggregates are formed using *partitioning functions*. 2^{nd} order locations are locations which contain a full 1^{st} order graph as their process, and the skeleton of that graph as their sort.

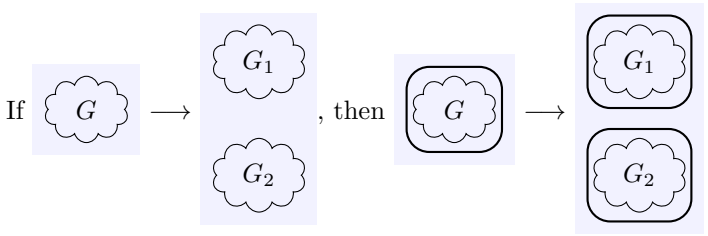


Figure 3.3: Intuition of the first candidate for $\mathcal{T}_u$: if a graph G can take a transition to become G', then $n(G)$ can take a transition to become $n(G')$. Notice that, in the 1st order graph, the transition is a *location graph transition* (from a graph to an other graph, with authorisation checking), while the transition in the 2nd order graph is an unconstrained location transition (from a location to a graph, without any authorisation checks).

We showed that nesting is reversible, in the sense that we can extract the original graph out of its location, and this scales to full graphs.

Finally, we also showed that the different ways to partition a graph form a lattice, which is an important point to remember for latter work.

3.3.2 Semantics of the 2nd order graph

We now focus on the dynamic aspects of nested location graphs. In this section, we present the semantics of 2nd order graphs. To define the semantics of the 2nd order graphs, we have to built a set $\mathcal{T}_2$ of unconstrained location transitions, and to define an authorisation function $\texttt{Auth}_2$. Our final goal is, of course, to show that the behaviour of the 1st order graph is preserved, via the exhibition of a partial bisimulation relation between the two.

Therefore, intuitively, the semantics of the 2nd order graph should mimic as much as possible the semantics of the 1st order graph. In the first part of this section, we aim to find a suitable candidate for unconstrained location transitions of the 2nd order graph. We illustrate problems that may occur with trivial candidates and we end up with a set of unconstrained location transition which is both easy to understand (as it is built on top of the most intuitive candidate), yet suitable for our future work.

In the second paragraph, we study the authorisation function of the 2nd order graph. The main idea is to use the authorisation function of the 1st order graph. The only difficulty of this part is to recover the 1st order skeleton graph which is used by that authorisation function. With that skeleton recovered, we can define the 2nd order authorisation function as a predicates which holds if and only if the nested 1st order graph can perform an analogous transition.

Unconstrained location transitions. Locations of the 2nd order graph are $[G : \Sigma(G) \lhd p \bullet r]$ where G is a 1st order graph. In order to study the semantics of 2nd order graphs, we first define their unconstrained transitions $\mathcal{T}_2$. We propose three candidates for $\mathcal{T}_2$, from the most intuitive to one that allows a bisimulation relation between the 1st and 2nd order graphs.

In the following $\mathcal{T}_1$ is the set of unconstrained location transition of the 1st order graph and $\mathcal{T}_2$ is the set of unconstrained location transition of the 2nd order graph.

First candidate. The most intuitive candidate for the set $\mathcal{T}_2$ of unconstrained transitions of the 2nd order graph would be to map each transition of the nested subgraph to a transition of the nesting location. Figure 3.3 shows the intuition of the first candidate set of $\mathcal{T}_2$. Formally, the set $\mathcal{T}_2$ is defined as follow:

$$\mathcal{T}_2 \triangleq \{\Delta_2 \cdot \emptyset \rhd n(G) \xrightarrow{\Lambda} N_p(G') | \Delta \cdot \Sigma \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} G' \wedge p \in \mathbb{P}\}$$

Remark. We do not go into further details of what is Δ_2, as the limitations developped below do not depend on it. ◁

However, as such, the transition relation of the 2nd order graph is not that interesting, as a transition can split the graph, but location can never merge on their own[9]. Therefore, using that semantics, in the general case, $\Gamma \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} G'$ does not necessarily implies $\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda_2} N_p(G')$ (for Γ_2 and Λ_2 properly adjusted[10] from Γ and Λ). The best achievement we could guarantee with such semantics would

[9] We would rely on semantics of the 1st order graph to remove and recreate all locations of one side to perform some kind of merging.

[10] According to dictionaries, I could write "suitably suitable" instead of "properly adjusted" here, which would be much funnier, though less understandable.

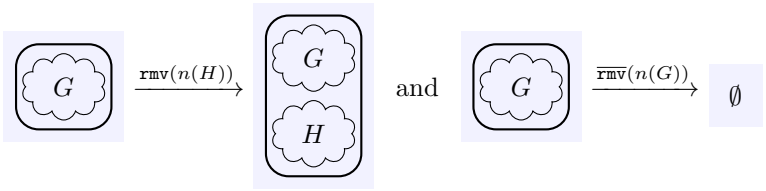
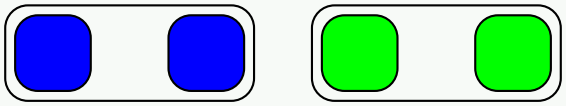


Figure 3.4: Intuition of the two administrative rules of the second candidate set. Notice that these two rules can only fire in parallel (due to the $\texttt{rmv}(n(H))$-$\overline{\texttt{rmv}}(n(H))$ tokens that should match). When fired, a location graph $n(G) \parallel n(H)$ reduces to $n(G \parallel H)$.

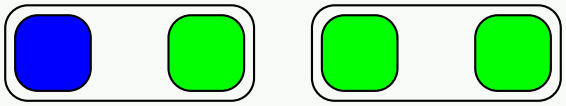
be that, for each $\Gamma \vdash_{\mathfrak{T}_1} G \xrightarrow{\Lambda} G'$, we would have $\Gamma_2 \vdash_{\mathfrak{T}_2} N_p(G) \xrightarrow{\Lambda_2} N_{p'}(G')$, with p' a finer partitioning function than p.

Intuitively, this candidate characterise systems in which new encapsulation scopes can be created (by splitting a agglomerate into multiple ones), but in which a location can not be exchanged among already existing encapsulation scopes.

Example 6 (Transition not captured)**.** Suppose we have a system with green and blue locations, and we separate them according to their color. In addition, suppose that location can change their color dynamically (i.e. they change the group they belong to).



At this point, if a location changes its color, the color change is also captured in the nested graph, but the aggregates remain unchanged: the first candidate rule does not allow the move of location across aggregates.



The goal of the second and third candidate rules is to provide a mechanism to allow such movements.

Second candidate. To overcome this problem, we could add two rules to allow the merging of locations. We call these rules *administrative*, as they are added only for problem solving, compared to the *proper* rule that does the actual work to mimic transitions of the 1st order graph. Intuitively, we want to merge two aggregates (in the case we would like to merge only part of an aggregate, we can already split it, then merge only the part we are interested in). We could therefore introduce two (sets of) synchronized rules: the first would capture an aggregate $n(H)$, and merge it with its already nested graph G, and therefore, $n(G)$ would become $n(G \parallel H)$; the second rule would simply be to allow a location $n(H)$ to remove itself. The intuition shows that, when firing both simultaneously, we would not loose any 1st order locations, and we could merge aggregates. Figure 3.4 shows the intuition for the two new administrative rules.

Formally, these two new administrative (sets of) rules would be the following:

— $\Delta \cdot \emptyset \rhd n(G) \xrightarrow{\langle \varepsilon, \{\hat{r}:\texttt{rmv}\langle n(H)\rangle\}\rangle} n(G \parallel H)$

— $\Delta \cdot \emptyset \rhd n(G) \xrightarrow{\langle \varepsilon, \{\hat{r}:\overline{\texttt{rmv}}\langle n(G)\rangle\}\rangle} \emptyset$

for any $r \in \mathbb{R}_2$, and adequates Δ. The new set $\mathfrak{T}_2$ would be the union of the first candidate and these two (sets of) rules.

The goal of these transitions is to manage the administrative transitions of the graph. However, we can anticipate a new problem that arises when adding these transitions: our (future) goal is to find a strong bisimulation between the 1st and 2nd order graphs —which means that each transition of one location graph should be matched by one and only one transition of the other location graph—, and such administrative transitions would not be matched (in the general case) by any transition of the 1st order graph. We can foresee that this second candidate will make our life[11] harder when it comes to prove the

[11] Well, mine at least.

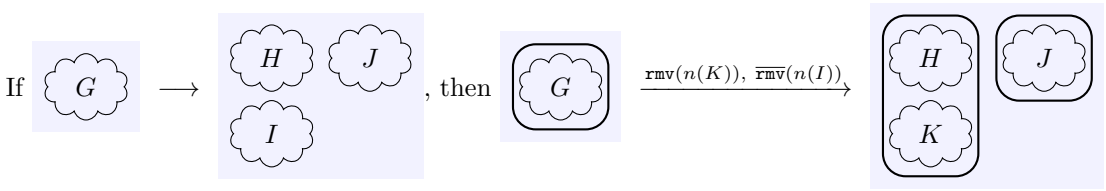


Figure 3.5: Intuition of the rule for the third candidate for the set of unconstrained location transitions of the 2nd order graph. In this rule, a 2nd order location $n(G)$ embedding a graph G (which reduces to G') can reduce to any location graph K, as long as (i) if a subgraph of G' is not nested in a location of K, then an appropriate $\overline{\texttt{rmv}}$ token should be put in the label (e.g. I in the figure); (ii) if K embeds locations that are not in G', then an appropriate $\texttt{rmv}$ token should be put in the label (e.g. K in the figure).

bisimulation.

Third candidate. The problem of the second candidate $\mathcal{T}_2$ is that administrative and proper rules are not taken in the same transition. An intuitive solution to the problem of the second candidate would therefore be to embed the administrative transitions with the proper transition. For that, we merge the behaviour of administrative rules within the rule of the first candidate candidate. Figure 3.5 shows the intuition of the rule.

Before actually writing down the set of rules, let first define what the new label should look like. Intuitively, we want to add (administrative) interactions: an $\hat{r} : \texttt{rmv}\langle n(L)\rangle$ for each new L merged, and an $\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle$ for each L removed.

We define the predicate $\texttt{inter}_2([\cdot],[\cdot],[\cdot],[\cdot])$ which is used to characterise correct interactions on 2nd order labels. Notice that this is not a function: we have the choice of $r \in \mathbb{R}_2$ when we add new administrative interactions. The semantics rules of location graphs constraint those roles to be unbound by the 2nd order location on which the transition applies.

Definition 57 (Second order interactions)**.** The predicate $\texttt{inter}_2(\iota, H, G, \iota_c)$ is defined according to the following inference rules.

$$\frac{\iota = \iota_c \qquad H \equiv G}{\texttt{inter}_2(\iota, H, G, \iota_c)} \qquad \frac{\texttt{inter}_2(\iota \cup \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}, H, G', \iota_c) \qquad G \equiv G' \parallel L \qquad L \notin H \qquad r \in \mathbb{R}_2}{\texttt{inter}_2(\iota, H, G, \iota_c)}$$

$$\frac{\texttt{inter}_2(\iota \cup \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}, H', G, \iota_c) \qquad H \equiv H' \parallel L \qquad L \notin G \qquad r \in \mathbb{R}_2}{\texttt{inter}_2(\iota, H, G, \iota_c)}$$

Finally, unconstrained location transitions are those of the form: $\Gamma_2 \rhd n(G) \xrightarrow{\Lambda_2} N_p(H)$ such that $\Gamma \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} G'$; $\Lambda_2.\texttt{prior} = \Lambda.\texttt{prior}$; and $\texttt{inter}_2(\Lambda.\texttt{sync}, H, G', \Lambda_2.\texttt{sync})$, where $\Gamma_2 = \Delta_2 \cdot \emptyset$ is an environment such that: (i) $\Gamma.\texttt{names} \subseteq \Delta_2$; and (ii) $\Delta_2 - \Gamma.\texttt{names} \subset \mathbb{R}_2$. That is, the names should be the same than in the 1st order graph reduction, up to roles in $\mathbb{R}_2$.

We keep the second administrative rule of the second candidate, to allow some locations to merge even though they do not reduce:

$$\Gamma \rhd n(G) \xrightarrow{\langle \varepsilon, \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle \mid L \in G \wedge r \in \mathbb{R}_2\}\rangle} \emptyset$$

Why shall we keep that administrative rule? Suppose that we have a graph $G = G_a \parallel G_b$, and the partitioning function we want is p. Suppose, in addition, that $p(G) = \{G_a, G_b\}$. In that case, $N_p(G) = n(G_a) \parallel n(G_b)$. Of course, if $\Gamma \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} G'$, we want $\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda_2} N_p(G')$.
Now suppose the only transition is a (Ctx) transition, where $\Gamma \vdash_{\mathcal{T}_1} G_a \xrightarrow{\Lambda_a} G'_a$ and (therefore) $\Gamma \vdash_{\mathcal{T}_1} G_a \parallel G_b \xrightarrow{\Lambda} G'_a \parallel G_b$, and that there is no transition for G_b.
In that case, we seek a 2nd order transition such that $\Gamma_2 \vdash_{\mathcal{T}_2} n(G_a) \parallel n(G_b) \xrightarrow{\Lambda_2} N_p(G'_a \parallel G_b)$.
Finally, suppose that $p(G'_a \parallel G_b) = \{G'_a \parallel G_b\}$. In that case, our 2nd order transition should yield $n(G'_a \parallel G_b)$.
Of course, $\Gamma_2 \rhd n(G_a) \xrightarrow{\Lambda_{2a}} n(G'_a \parallel G_b)$ is in $\mathcal{T}_2$ (for an adequate Λ_{2a}). But without that additional administrative rule, $n(G_b)$ can not self remove, since there is no 1st order reduction for G_b, and

therefore, there is no possibility to remove the $n(G_b)$ so that $n(G_a)$ can capture it.
With that additional administrative rule, it is completely different: with $\Gamma_2 \rhd G_b \xrightarrow{\Lambda_{2b}} \emptyset$, the 2nd order graph can take a (Comp) transition (assuming all other premisses are satisfied)[a]:

$$\Gamma_2 \vdash_{\mathcal{T}_2} n(G_a) \parallel n(G_b) \xrightarrow{\Lambda} n(G'_a \parallel G_b)$$

Notice that, when such things happens, a (Ctx) transition of the 1st order graph is matched by a (Comp) transition.

[a] The label of that transition is indeed the original Λ: all added interactions are matched. We will give further details on that in the bisimulation proof.

Finally, our transition set $\mathcal{T}_2$ is the union of these rules:

Definition 58 (2nd order unconstrained location transitions)**.**

$$\mathcal{T}_2 \triangleq \{\Delta_2 \cdot \emptyset \rhd n(G) \xrightarrow{\langle \pi, \iota_2 \rangle} N_p(H) | \Delta \cdot G_s \vdash_{\mathcal{T}_u} G \xrightarrow{\langle \pi, \iota_1 \rangle} G' \wedge \texttt{inter}_2(\iota_1, H, G', \iota_2) \wedge \Delta_2 = \Delta \cup \Delta_r \wedge \Delta_r \subset \mathbb{R}_2\}$$
$$\cup \{\Gamma \rhd n(G) \xrightarrow{\langle \varepsilon, \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle | L \in G \wedge r \in \mathbb{R}_2\}\rangle} \emptyset\}$$

Authorisation function. We now need to build the authorisation function of the 2nd order graph. The idea is to recover the underlying 1st order transition, and to check (using the 1st order authorisation function) if it is allowed.

To achieve that, we need three steps: (i) we need to extract the 1st order skeleton graph corresponding to the current graph; (ii) we need to retrieve the 1st order label of the underlying transition; and (iii) we need to find the 1st order *destination* graph (since locations can be added or removed, according to the unconstrained location transitions set).

Let $e : \Sigma(\mathbb{G}_2) \to \Sigma(\mathbb{G})$ be a function that extracts the original 1st order skeleton graph from a 2nd order skeleton graph:

Definition 59 (Skeleton extraction)**.**

$$e(G_s) = \prod_{L_s \in G_s} L_s.\texttt{sort}$$

This extraction process is sound, in the sens that it indeed returns the skeleton of the underlying 1st order graph.

Lemma 19 (Extraction soundness)**.**

$$\forall G \in \mathbb{G} \cdot \forall p \in \mathbb{P}_{\mathbb{G}} \cdot e(\Sigma(N_p(G))) = \Sigma(G)$$

Proof.

$$e(\Sigma(N_p(G))) = \prod_{L_s \in \Sigma(N_p(G))} L_s.\texttt{sort}$$
$$\overset{\text{Lem 2}}{=} \prod_{L_s \in \prod_{L \in N_p(G)} \Sigma(L)} L_s.\texttt{sort} = \prod_{L \in N_p(G)} \Sigma(L).\texttt{sort}$$
$$\overset{\text{Def 55}}{=} \prod_{L \in \prod_{G_i \in p(G)} n(G_i)} \Sigma(L).\texttt{sort} = \prod_{G_i \in p(G)} \Sigma(n(G_i)).\texttt{sort}$$
$$\overset{\text{Def 49}}{=} \prod_{G_i \in p(G)} \Sigma([G_i : \Sigma(G_i) \lhd p \bullet r]).\texttt{sort}$$
$$\overset{\text{Def 15}}{=} \prod_{G_i \in p(G)} [\Sigma(G_i) \lhd p \bullet r].\texttt{sort} = \prod_{G_i \in p(G)} \Sigma(G_i)$$
$$\overset{\text{Lem 3, Def 51}}{=} \Sigma(G)$$

□

Our second step is to retrieve the 1st order label corresponding to the 2nd order transition we are authorizing. Fortunately, we introduced roles in $\mathbb{R}_2$ specifically for administrative purposes. Therefore, to retrieve the original label, we can simply prune our label of all elements on roles in $\mathbb{R}_2$. We define a pruning function that removes roles in $\mathbb{R}_2$.

Definition 60 (Label pruning)**.**

$$\texttt{prune}(\Lambda) = \langle \{\hat{r} : *a\langle V\rangle | \hat{r} : *a\langle V\rangle \in \Lambda.\texttt{prior} \wedge r \in \mathbb{R}\}, \{\hat{r} : \hat{a}\langle V\rangle | \hat{r} : \hat{a}\langle V\rangle \in \Lambda.\texttt{sync} \wedge r \in \mathbb{R}\}\rangle$$

Lemma 20 (Pruned label lemma)**.** *For any Δ, G_2, Λ and G_2', if $\texttt{prune}(\Lambda) = \Lambda$ and $\Delta \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} G_2' \in \mathcal{T}_2$, then there exists G_1^s, G_1, G_1' such that*

(i) $\Delta \cdot G_1^s \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda} G_1'$*; and*
(ii) $G_2 = n(G_1)$*; and*
(iii) $G_2' = N_p(G_1')$*.*

Proof. By hypothesis, $\Delta \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} G_2' \in \mathcal{T}_2$. By definition of $\mathcal{T}_2$, either: (i) $G_2 = n(G_1)$, $G_2' = N_p(H)$ for some G_1 and H two first order graphs such that $\Gamma \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} G_1'$, with $\texttt{inter}_2(\Lambda_1.\texttt{sync}, H, G', \Lambda.\texttt{sync})$; or (ii) $G_2' = \emptyset$ and $\Lambda = \langle \varepsilon, \{\hat{r} : \overline{\texttt{rmv}}(n(G))\}\rangle$ with $r \in \mathbb{R}_2$ Since $\texttt{prune}(\Lambda) = \Lambda$, (ii) is not possible.
Concerning the possibility (i), since $\texttt{inter}_2(\Lambda_1.\texttt{sync}, H, G', \Lambda.\texttt{sync})$, and $\texttt{prune}(\Lambda) = \Lambda$, then $\Lambda_1.\texttt{sync} = \Lambda.\texttt{sync}$ (hence $\Lambda_1 = \Lambda$) and $H = G'$.
Thus, $\Gamma \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda} G_1'$, for an adequate Γ. □

The third step is to retrieve the original 1st order destination graph of the underlying 1st order transition. If to authorize a transition $\Delta \cdot \emptyset \rhd L_2 \xrightarrow{\Lambda_2} C$, we would like to know the G_1 such that $C = N_p(G)$. However, Λ_2 can possibly contains $\hat{r} : \texttt{rmv}\langle n(L_1)\rangle$ or $\hat{r} : \overline{\texttt{rmv}}\langle n(L_1)\rangle$, corresponding to added or removed locations of G. Therefore, to recover the original 1st order G, we have to *revert* those actions of Λ_2.

Definition 61 (Graph recovery)**.**

$$\texttt{recover}(\Lambda, G_2) = \begin{cases} \texttt{recover}(\Lambda', N_p(N_p^{-1}(G_2) \parallel L)) & \text{if } \Lambda = \langle \Lambda'.\texttt{prior}, \Lambda'.\texttt{sync} \cup \{\hat{r} : \overline{\texttt{rmv}}\langle L\rangle\}\rangle \\ \texttt{recover}(\Lambda', N_p(G_1)) & \text{if } G_2 = N_p(G_1 \parallel L) \\ & \quad \wedge \Lambda = \langle \Lambda'.\texttt{prior}, \Lambda'.\texttt{sync} \cup \{\hat{r} : \texttt{rmv}\langle L\rangle\}\rangle \\ G_2 & \text{otherwise} \end{cases}$$

Finally, the authorisation function of the 2nd order graph allows transitions which correspond to 1st order allowed transitions, given the right environment:

Definition 62 (2nd order authorisation function)**.**

$$\texttt{Auth}(G_s^2, \Delta \cdot \emptyset \rhd L_2 \xrightarrow{\Lambda_2} C) \Leftrightarrow \Delta' \cdot e(G_s^2) \vdash_{\mathcal{T}_1} n^{-1}(L_2) \xrightarrow{\texttt{prune}(\Lambda_2)} N_p^{-1}(\texttt{recover}(\Lambda_2, C))$$

with $\Delta' = \Delta \cap \mathbb{R}$ (i.e. Δ without out-of-band roles).

Remark. The authorisation function of the 2nd order graph does not depend on the chosen partition function. ◁

2nd order semantic lemmas. In this paragraph, we introduce an important result on the 2nd order semantics. This result states that the partition function is not that relevant, in that we can change from one to the other during reduction.

The proof of this statement is done in two steps: (i) in a first step, we show in Lemma 21 that when we perform a transition with a partitionning function p_1, we can also perform a transition to reach a coarser partition p_2 (intuitively, during a transition, we can merge together some nested aggregates); then (ii) we

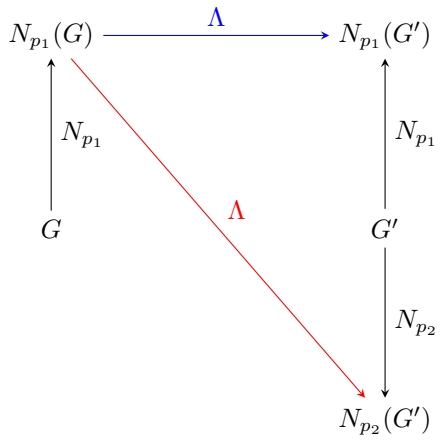


Figure 3.6: Illustration of Lemma 23. Given a 2nd order graph $N_{p_1}(G)$ which reduces to $N_{p_1}(G')$ (the blue arrow), we show that $N_{p_1}(G)$ also reduces to $N_{p_2}(G')$ (the red arrow) for any p_2.

show the reverse in Lemma 22, that is, we can always reach a p_2 which is finer than p_1 (intuitively, during a transition, we can split some aggregates), and from that, it follows that we can reach any partition (Lemma 23).

Figure 3.6 gives a graphical interpretation of this lemma.

Lemma 21. *For any graphs G and G', for all partition functions p, p_1 and p_2 such that $p_1(G') \succ p_2(G')$, for any environment Γ, if $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_1}(G')$, then*

$$\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_2}(G')$$

Proof. By hypothesis, $p_1(G') \succ p_2(G')$. Therefore, from Lemma 12, $\forall G_i \in p_1(G'), \exists G'_i \in p_2(G'), G_i \subseteq G'_i$. Without loss of generality, consider there are only two elements of $p_1(G')$ (named G_i and G_j hereafter) such that $G_i \in p_1(G')$, $G_j \in p_1(G')$ and $G_i \parallel G_j \in p_2(G')$. For all other elements $G_k \in (p_1(G') - \{G_i, G_j\})$, $G_k \in p_2(G')$. Let $G^2_k = \prod_{G_k \in p_1(G') - \{G_i, G_j\}} n(G_k)$, $G^2_i = n(G_i)$, $G^2_j = n(G_j)$ and $G^2_{ij} = n(G_i \parallel G_j)$.

The goal of the proof is to show that, given $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} G^2_i \parallel G^2_j \parallel G^2_k$, then $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} G^2_{ij} \parallel G^2_k$.

By case analysis of $p(G)$:

Case $|p(G)| = 1$: In this case $N_p(G) = n(G)$, and $\Gamma \vdash n(G) \xrightarrow{\Lambda} n(G_i) \parallel n(G_j) \parallel G_k$, using (Trans). From the premisses of (Trans), $\Gamma \rhd n(G) \xrightarrow{\Lambda} n(G_i) \parallel n(G_j) \parallel G_k$ and the authorisation function holds. Since the authorisation function does not depend on the chosen partition function, we only need to show that $\Gamma \rhd n(G) \xrightarrow{\Lambda} n(G_i \parallel G_j) \parallel G_k$, which holds directly from the definition of the unconstrained location transitions of the 2nd order graph.

Case $|p(G)| = n + 1$: To clarify the proof, we distinguish the case $|p(G)| = 2$ (proven using the case $|p(G)| = 1$) and the case $|p(G)| > 2$ (proven using the cases $|p(G)| = 1$ and $|p(G)| = 2$).

Case $|p(G)| = 2$: We have $N_p(G) \equiv G_a \parallel G_b$ and either rule (i) (Comp); or (ii) (Ctx) applies.

Case (Comp): We have $\Gamma \vdash G_a \xrightarrow{\Lambda} G'_a$ and $\Gamma \vdash G_b \xrightarrow{\Lambda} G'_b$.

Case $G'_a \equiv G^2_i \parallel G_{ka}$ and $G'_b \equiv G^2_j \parallel G_{kb}$ (resp. vice-versa): We have $\Gamma \rhd G_a \xrightarrow{\Lambda_a} G'_a$ and $\Gamma \rhd G_b \xrightarrow{\Lambda_b} G'_b$ with $\texttt{seval}(\Lambda_a \cup \Lambda_b) = \Lambda$. Thus, the rules $\Gamma \rhd G_a \xrightarrow{\Lambda_a \cup \langle \varepsilon, \hat{r}:\texttt{rmv}\langle G^2_j \rangle\rangle} G^2_{ij} \parallel G_{ka}$ and $\Gamma \rhd G_b \xrightarrow{\Lambda_b \cup \langle \varepsilon, \hat{r}:\overline{\texttt{rmv}}\langle G^2_j \rangle\rangle} G_{kb}$ hold. Finally, with rule (Comp), $\Gamma \vdash G_a \parallel G_b \xrightarrow{\Lambda} G^2_{ij} \parallel G_k$.

Case $G'_a \equiv G^2_i \parallel G^2_j \parallel G_{ka}$ (resp. for G'_b): The result holds directly from the induction hypothesis on G_a (resp. G_b).

Case (Ctx): By symmetry, suppose that $\Gamma \vdash G_a \xrightarrow{\Lambda} G'_a$ and G_b does not reduce.

Case $G_b = G_j^2$ (resp. $G_b = G_i^2$): G_b can take the transition $\Gamma \vdash G_b \xrightarrow{\langle \varepsilon, \hat{r}:\overline{\texttt{rmv}}\langle G_b \rangle\rangle} \emptyset$. In addition, $G'_a \equiv G_i^2 \parallel G_k$ (resp $G_a\prime \equiv G_j^2 \parallel G_k$). Thus, G_a can take the transition $\Gamma \vdash G_a \xrightarrow{\Lambda \cup \langle \varepsilon, \hat{r}:\texttt{rmv}\langle G_b \rangle\rangle} G_{ij}^2 \parallel G_k$. Finally, with rule (Comp), $\Gamma \vdash G_a \parallel G_b \xrightarrow{\Lambda} G_{ij}^2 \parallel G_k$.

Case $G_b \neq G_i^2$ and $G_b \neq G_j^2$: The result holds directly from the induction hypothesis on G_a.

Case $|p(G)| > 2$: We have to cases to distinguish the case in which (i) both G_i^2 and G_j^2 exist in $N_p(G)$; or (ii) every other possibilities.

Case $N_p(G) \equiv (G_i^2 \parallel G_j^2) \parallel N_{p'}(G')$ and rule (Ctx) applies: We have $\Gamma \vdash N_{p'}(G') \xrightarrow{\Lambda} G_k^2$. Let $G_r \in N_{p'}(G')$ be a location that takes a (Trans) reduction. Hence, $\Gamma \rhd G_r \xrightarrow{\Lambda_r} G'_r$, therefore $\Gamma \rhd G_r \xrightarrow{\Lambda_r} G'_r \parallel G_{ij}^2$ is a valid unconstrained location reduction. In addition, from the definition of the unconstrained location transitions, both G_i^2 and G_j^2 can reduce: $\Gamma \rhd G_i^2 \xrightarrow{\langle \varepsilon, \hat{r_1}:\overline{\texttt{rmv}}\langle G_i^2 \rangle\rangle} \emptyset$ and $\Gamma \rhd G_j^2 \xrightarrow{\langle \varepsilon, \hat{r_2}:\overline{\texttt{rmv}}\langle G_j^2 \rangle\rangle} \emptyset$. Thus, $\Gamma \vdash G_i^2 \parallel G_j^2 \parallel G_r \xrightarrow{\Lambda_r} G'_r \parallel G_{ij}^2$. Therefore, finally, $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} G_{ij}^2 \parallel G_k^2$.

Otherwise: The result follows directly from the cases $|p(G)| = 1$ or $|p(G)| = 2$.

□

Lemma 22. *For any graphs G and G', for all partition functions p_1 and p_2 such that $p_2(G') \succ p_1(G')$, for any environment Γ, if $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_1}(G')$, then*

$$\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_2}(G')$$

Proof. Without loss of generality, consider there are a two elements of $p_2(G')$ (named G_i and G_j hereafter) such that $G_i \in p_2(G')$, $G_j \in p_2(G')$ and $G_i \parallel G_j \in p_1(G)$. For all other elements $G_k \in p_2(G')$, $G_k \in p_2(G') - \{G_i, G_j\}$.
By induction on $|p(G)|$:

Case $|p(G)| = 1$: $N_p(G) = n(G)$ and $\Gamma \rhd n(G) \xrightarrow{\Lambda} N_{p_1}(G')$. We have $N_{p_1}(G') = G_{ij} \parallel G_k$. We have, from the definition of the unconstrained location transitions, that $\Gamma \rhd n(G) \xrightarrow{\Lambda} N_{p_2}(G')$ is also a valid unconstrained location transitions, which ends this case.

Case $|p(G)| = n + 1$: In this case, either (i) (Comp); or (ii) (Ctx) applies.
Case (Comp): The results follows directly from the induction hypothesis.

Case (Ctx): We have $N_p(G) \equiv G_a \parallel G_b$. By symmetry, consider that $\Gamma \vdash G_a \xrightarrow{\Lambda} G'_a$ and that G_b does not reduce. We have $N_{p_1}(G') \equiv G'_a \parallel G_b$.
We distinguish two cases:
Case $G_{ij} \in G'_a$: The result follows directly from the induction hypothesis.

Case $G_{ij} \in G_b$: In this case, $G_b \equiv G_{ij} \parallel G'_b$, which reduces (using (Trans) on G_b and (Ctx)) as $\Gamma \vdash G_b \xrightarrow{\langle \varepsilon, \hat{r}:\overline{\texttt{rmv}}\langle G_{ij} \rangle\rangle} G'_b$. Also, since $\Gamma \vdash G_a \xrightarrow{\Lambda} G'_a$, there is at least one location $G_r \in G_a$ such that $\Gamma \vdash G_r \xrightarrow{\Lambda_r} G'_r$ using (Trans). Thus, G_r also reduces as $\Gamma \vdash G_r \xrightarrow{\Lambda \cup \langle \varepsilon, \hat{r}:\texttt{rmv}\langle G_{ij} \rangle\rangle} G_i \parallel G_j \parallel G'_r$. Finally, using (Comp), $\Gamma \vdash G_a \parallel G_b \xrightarrow{\Lambda} G'_a \parallel G_i \parallel G_j \parallel G'_b$.

□

Lemma 23 (2^{nd} order semantic lemma)**.** *For all partition functions p, p_1 and p_2, for any environment Γ, for any graphs G and G', if $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_1}(G')$, then*

$$\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_2}(G')$$

Proof. From Corollary 1, the partitions of G' form a lattice with respect to $\succ$. Let $p_{\text{inf}} = \inf(p_1, p_2)$[a]. By definition of inf, $p_1 \succ p_{\text{inf}}$ and $p_2 \succ p_{\text{inf}}$.
From Lemma 21, $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_{\text{inf}}}(G')$.
From Lemma 22, $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_{p_2}(G')$. □

[a]The partition p_{inf} is guaranteed to exist since $\langle \mathbb{P}_{G'}, \succ \rangle$ is a lattice.

An other formulation of this lemma is the following:

Corollary 3 (Interchangeable partition functions). For any partition functions p_1 and p_2, for any label Λ,

$$\Gamma \vdash_{\mathcal{T}_2} G \xrightarrow{\Lambda} N_{p_1}(C) \Leftrightarrow \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} N_{p_2}(C)$$

Corollary 4 (of 2nd order semantic lemma). For any partition function p, for any environment Γ and for any graphs G, G', if $\Gamma \vdash N_p(G) \xrightarrow{\Lambda} N_p(G_1) \parallel N_p(G_2)$, then

$$\Gamma \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_p(G_1 \parallel G_2)$$

3.3.3 Partial bisimulation

In this subsection, we show that our nesting preserves the behaviour of the location graph. We use the notion of $\rightleftharpoons$-bisimulation we defined above. In the first paragraph, we define the equivalence (the relation $\rightleftharpoons$) we will be using. Then, in the second paragraph, we show the bisimulation result.

Environment & label equivalences. Intuitively, if λ_1 is a 1st order label and λ_2 a 2nd order label, λ_1 is equivalent to λ_2 if actions of λ_2 are either out-of-band, or in λ_1.

Definition 63 (Label equivalence for nested location graphs)**.**

$$\rightleftharpoons_\lambda \stackrel{\Delta}{=} \{\langle \lambda_1, \lambda_2 \rangle | \forall a \in \lambda_2.\texttt{sync} \cdot (a \in \lambda_1.\texttt{sync} \vee a.\texttt{role} \in \mathbb{R}_2) \wedge (\lambda_1.\texttt{prior} = \lambda_2.\texttt{prior})\}$$

Concerning environments, a 1st order environment $\Delta_1 \cdot G_1^s$ is equivalent to a 2nd order environment $\Delta_2 \cdot G_2^s$ if the 1st order skeleton graph embedded in the sorts of the skeleton locations of G_2^s forms G_1^s.

Definition 64 (Skeleton graph equivalence for nested location graphs)**.**

$$\rightleftharpoons_s \stackrel{\Delta}{=} \{\langle G_1^s, G_2^s \rangle | G_1^s = e(G_2^s)\}$$

Concerning atoms, we consider that two sets of names are equivalent if the second one contains some additionnal roles from $\mathbb{R}_2$.

Definition 65 (Atom equivalence for nested location graphs)**.**

$$\rightleftharpoons_\Delta \stackrel{\Delta}{=} \{\langle \Delta_1, \Delta_2 \rangle | \Delta_2 = \Delta_1 \cup \Delta_r \wedge \Delta_r \subseteq \mathbb{R}_2\}$$

Finally, two environments are equivalent if their atoms and their skeleton graphs are equivalent.

Definition 66.

$$\rightleftharpoons_E \stackrel{\Delta}{=} \{\langle \Delta_1 \cdot G_1^s, \Delta_2 \cdot G_2^s \rangle | \Delta_1 \rightleftharpoons_\Delta \Delta_2 \wedge G_1^s \rightleftharpoons_s G_2^s\}$$

In this very case, an interesting point to note is that for all G, $\texttt{supp}(G) \cdot \Sigma(G) \rightleftharpoons_E \texttt{supp}(N_p(G)) \cdot \Sigma(N_p(G))$.

Lemma 24.

$$\forall G \cdot \forall p \cdot \texttt{supp}(G) \cdot \Sigma(G) \rightleftharpoons_E \texttt{supp}(N_p(G)) \cdot \Sigma(N_p(G))$$

Proof. There are two points to prove: (i) $\mathtt{supp}(G) \asymp_\Delta \mathtt{supp}(N_p(G))$; and (ii) $\Sigma(G) \asymp_s \Sigma(N_p(G))$.
Item (i): By induction on $p(G)$:

Case $G = \emptyset$ ($p(G) = \emptyset$): Trivial, $\mathtt{supp}(G) = \mathtt{supp}(N_p(G)) = \emptyset$.

Case $p(G) = \{G\}$: $N_p(G) = n(G) = [G : \Sigma(G) \lhd p \cup p_2 \bullet r \cup r_2]$, with $p_2 \subset \mathbb{R}_2$ and $r_2 \subset \mathbb{R}_2$.
Therefore, $\mathtt{supp}(N_p(G)) = \mathtt{supp}(G) \cup \mathtt{supp}(\Sigma(G)) \cup \mathtt{supp}(p) \cup \mathtt{supp}(p_2) \cup \mathtt{supp}(r) \cup \mathtt{supp}(r_2) = \mathtt{supp}(G) \cup \mathtt{supp}(p_2) \cup \mathtt{supp}(r_2)$; and $\mathtt{supp}(p_2) \cup \mathtt{supp}(r_2) \subset \mathbb{R}_2$.

Case $|p(G)| > 1$: $\mathtt{supp}(N_p(G)) = \mathtt{supp}(\prod_{G_i \in p(G)} n(G_i)) = \bigcup_{G_i \in p(G)} \mathtt{supp}(n(G_i))$.
From our induction hypothesis, we know that, for each G_i, $\mathtt{supp}(n(G_i)) = \mathtt{supp}(G_i) \cup \Delta_r^i$, with $\Delta_r^i \subset \mathbb{R}_2$.
Therefore, $\bigcup_{G_i \in p(G)} \mathtt{supp}(n(G_i)) = \bigcup_{G_i \in p(G)} \mathtt{supp}(G_i) \cup \Delta_r^i = (\bigcup_{G_i \in p(G)} \mathtt{supp}(G_i)) \cup (\bigcup_{G_i \in p(G)} \Delta_r^i) = \mathtt{supp}(G) \cup \bigcup_{G_i \in p(G)} \Delta_r^i$; and, of course, $\bigcup_{G_i \in p(G)} \Delta_r^i \subset \mathbb{R}_2$.
Therefore,

$$\forall G \in \mathbb{G} \cdot \mathtt{supp}(G) \asymp_\Delta \mathtt{supp}(N_p(G))$$

Item (ii): The proposition is a direct consequence of Lemma 19: $e(\Sigma(N_p(G))) = \Sigma(G)$.

□

Simulations. We now show that there is a simulation relation between G and $N_p(G)$. Let $\mathcal{S} \triangleq \{\langle G, N_p(G)\rangle | p \in \mathbb{P}^{\mathbb{G}_1}\}$.

Theorem 2 (The 2nd order graph simulates the 1st order graph)**.** *$\mathcal{S}$ is a $\langle \asymp_E, \asymp_\lambda \rangle$-simulation.*

Proof. We have two show the five items of Definition 46.
Item (i) is proved directly: for each $\langle G, N_p(G)\rangle \in \mathcal{S}$, if $\Delta_1 \cdot G_1^s \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G'$ then $\Sigma(G) \subseteq G_1^s$. Therefore $G_1^s \equiv \Sigma(G) \parallel \Sigma(G')$ for some G'. Then, we have that $e(\Sigma(N_p(G)) \parallel \Sigma(N_p(G'))) = \Sigma(G) \parallel \Sigma(G') = G_1^s$, therefore $G_1^s \asymp_s \Sigma(N_p(G)) \parallel \Sigma(N_p(G'))$.
From lemma 24, item (iv) implies item (v).
We show by induction that, for any 1st order graph G, for any partitioning function p, for any environment Γ, if $\Gamma \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} C$, then for any Γ_2 such that $\Gamma_1 \asymp_E \Gamma_2$, $\Sigma(N_p(G)) \subset \pi_2(\Gamma_2)$, and $\mathtt{supp}(N_p(G)) \subset \pi_1(\Gamma_2)$, we have: $\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_p(C)$. This statement shows items (iii) and (iv). In addition, since $\Lambda \asymp_\lambda \Lambda$, it also shows item (ii).
By induction on $|p(G)|^a$:
Base case ($|p(G)| = 1$): Since $|p(G)| = 1$, $N_p(G) = n(G)$.
We have, by hypothesis $\Gamma_1 \asymp_E \Gamma_2$. Let $\Gamma_1 = \Delta_1 \cdot G_1^s$ and $\Gamma_2 = \Delta_2 \cdot G_2^s$.
By definition of $\asymp_E$, $\Delta_2 = \Delta_1 \cup \Delta_2'$ with $\Delta_2' \subset \mathbb{R}_2$, and $G_1^s = e(G_2^s)$.
Thus, according to the definition of the 2nd order semantics, $\Delta_2 \cdot \emptyset \rhd n(G) \xrightarrow{\Lambda} N_p(C) \in \mathcal{T}_2$.
In addition, by hypothesis:

$$\Gamma_1 \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} C$$

Therefore, for all Λ_2 such that $\mathtt{prune}(\Lambda_2) = \Lambda$

$$\Delta_1 \cdot G_1^s \vdash_{\mathcal{T}_1} G \xrightarrow{\mathtt{prune}(\Lambda_2)} C$$
$$\Leftrightarrow \Delta_1 \cdot e(G_2^s) \vdash_{\mathcal{T}_1} n^{-1}(n(G)) \xrightarrow{\mathtt{prune}(\Lambda_2)} N_p^{-1}(N_p(C))$$

Since $\Delta_1 = (\Delta_1 \cup \Delta_2') \cap \mathbb{R} = \Delta_2 \cap \mathbb{R}$, by definition of the authorisation function for 2nd order graphs: $\mathtt{Auth}_2(G_2^s, \Delta_2 \cdot \emptyset \rhd n(G) \xrightarrow{\Lambda_2} N_p(C))$ holds. In particular, it holds for $\Lambda_2 = \Lambda$. Hence, $\Delta_2 \cdot N_p(G_s^1) \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_p(C)$ with rule (Trans).

Inductive case ($|p(G)| = n + 1$): Without loss of generality, consider that the partition $p(G)$ contains only two elements G_a and G_b. By definition[b] of $p(G)$, we have that $G \equiv G_a \parallel G_b$, i.e. $N_p(G)$ contains only two locations $n(G_a) \parallel n(G_b)$. Also, we assume that G reduces using the rule (Comp). It can reduce using (Ctx) and the proof is similar.
By hypothesis, $\Gamma_1 \vdash_{\mathcal{T}_1} G \xrightarrow{\Lambda} C$. Hence, after Lemma 10 in [33], $\Gamma_1 \vdash_{\mathcal{T}_1} G_a \parallel G_b \xrightarrow{\Lambda} G_a' \parallel G_b'$ with $C \equiv G_a' \parallel G_b'$. From the premisses of (Comp):

(i) $\Gamma_1 \vdash_{\mathcal{T}_1} G_a \xrightarrow{\Lambda_a} G'_a$; (ii) $\Gamma_1 \vdash_{\mathcal{T}_1} G_b \xrightarrow{\Lambda_b} G'_b$; (iii) $\Lambda.\texttt{sync} = \texttt{seval}(\Lambda_a.\texttt{sync}, \Lambda_b.\texttt{sync})$; and (iv) $\Lambda.\texttt{sync}.\texttt{roles} \subseteq G.\texttt{unbound}$.
Since $\Sigma(N_p(G)) \subseteq \pi_2(\Gamma_2)$, $\Sigma(n(G_a)) \in \pi_2(\Gamma_2)$ and $\Sigma(n(G_b)) \in \pi_2(\Gamma_2)$.

Using the base case[c], we know that $\Gamma_2 \vdash_{\mathcal{T}_2} n(G_a) \xrightarrow{\Lambda_a} N_{p'}(G'_a)$ (resp. for G_b), for a given p'[d].
We have to show that $\texttt{Cond}_I(\Lambda.\texttt{sync}, \Lambda_a.\texttt{sync}, \Lambda_b.\texttt{sync}, N_{p'}(G_a) \parallel N_{p'}(G_b))$ and $\texttt{Cond}(\Gamma_2, N_{p'}(G_a) \parallel N_{p'}(G_b))$ hold, so that we can deduce that $N_p(G)$ can take a (Comp) transition.
We know that $\Lambda.\texttt{sync} = \texttt{seval}(\Lambda_a.\texttt{sync})\Lambda_b.\texttt{sync}$. Also, $G.\texttt{unbound} \subseteq N_p(G).\texttt{unbound}$, from Lemma 18. Therefore, $\texttt{Cond}_I(\Lambda.\texttt{sync}, \Lambda_a.\texttt{sync}, \Lambda_b.\texttt{sync}, N_{p'}(G_a) \parallel N_{p'}(G_b))$ holds.
By hypothesis, $\Sigma(N_p(G)) \subset \pi_2(\Gamma_2)$ and $\texttt{supp}(N_p(G)) \subset \pi_1(\Gamma_2)$.
Also, from the induction hypothesis, we can directly deduce that $\texttt{Cond}_P(s, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma_2, N_{p'}(G_a), N_{p'}(G_b))$.
Hence, $\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_{p'}(G'_a) \parallel N_{p'}(G'_b)$ using (Comp).
After Lemma 23, $\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_p(G'_a \parallel G'_b)$. and finally , since $G'_a \parallel G'_b \equiv C$:

$$\Gamma_2 \vdash_{\mathcal{T}_2} N_p(G) \xrightarrow{\Lambda} N_p(C)$$

□

[a] Notice that we can not do an induction on the depth of the graph as usual, since we use the structural congruence in the inductive case, which can create deeper subgraphs, on which we could not apply our induction hypothesis; using the number of location instead does not suffer this problem.
[b] Definition 52.
[c] Since $G = G_a \parallel G_b$, then the number of location in G_a and G_b is at most n.
[d] Notice that p' is not necessarily the same than p, but it does not matter since Lemma 23 allows us to target any partitioning function.

The other direction is a bit more involved. We first prove two helpful lemmas, which allows us to relate actions on $\mathbb{R}_2$ and the addition and removal of locations. Essentially, the first lemma say that, if a transition *absorb* a new location L (via the synchronisation of an interaction $i = \hat{r} : \texttt{rmv}\langle n(L)\rangle$), then there exist a similar transition, which does not synchronise on i, and which does not absorb L.

The second lemma is the complement: it provides a similar result, for interactions $\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle$.

Lemma 25. *For any Γ, 2^{nd} order graph $G_2 = N_p(G_1)$ and $G'_2 = N_p(G'_1)$, $L \in \mathbb{L}$, Λ if $\hat{r} : \texttt{rmv}\langle n(L)\rangle \in \Lambda.\texttt{sync}$, $\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} G'_2$ and $r \in \mathbb{R}_2$ is in $G_2.\texttt{unbound}$, then:*

$$\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G''_1)$$

with $G'_1 \equiv G''_1 \parallel L$.

Proof. By induction on the reduction rule of $\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} G'_2$.
Case (Trans): In that case, $G_2 = N_p(G_1) = n(G_1)$. From the premisses of (Trans):

(i) $\Gamma.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} G'_2 \in \mathcal{T}_2$
(ii) $\Sigma(G_2) \in \Gamma.\texttt{graph}$
(iii) $\texttt{Auth}_2(\Gamma.\texttt{graph}, \Gamma.\texttt{names} \rhd G_2 \xrightarrow{\Lambda} G'_2)$

By definition of $\mathcal{T}_2$, since $\hat{r} : \texttt{rmv}\langle n(L)\rangle \in \Lambda.\texttt{sync}$, $G'_1 \equiv G''_1 \parallel L$, and $\Gamma \rhd G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G''_1) \in \mathcal{T}_2$.
By definition of $\texttt{Auth}_2$,

$$\begin{aligned}
&\texttt{Auth}_2(\Gamma.\texttt{graph}, \Gamma.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G''_1)) \\
\Leftrightarrow &(\Gamma.\texttt{names} \cap \mathbb{R}) \cdot e(\Gamma.\texttt{graph}) \vdash_{\mathcal{T}_1} n^{-1}(G_2) \xrightarrow{\texttt{prune}(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle)} \\
&\qquad N_p^{-1}(\texttt{recover}(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle, N_p(G''_1))) \\
\Leftrightarrow &(\Gamma.\texttt{names} \cap \mathbb{R}) \cdot e(\Gamma.\texttt{graph}) \vdash_{\mathcal{T}_1} n^{-1}(G_2) \xrightarrow{\texttt{prune}(\Lambda)} N_p^{-1}(\texttt{recover}(\Lambda, N_p(G''_1 \parallel L))) \\
\Leftrightarrow &\texttt{Auth}_2(\Gamma.\texttt{graph}, \Gamma.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} G'_2)
\end{aligned}$$

since $\texttt{prune}(\Lambda_2) = \texttt{prune}(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle)$.
Therefore $\texttt{Auth}_2(\Gamma.\texttt{graph}, \Gamma.\texttt{names} \rhd G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G_1''))$ holds.
Therefore,

$$\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G_1'')$$

Case (Comp): In that case, $G_2 = G_{2a} \parallel G_{2b}$, $G_2' = G_{2a}' \parallel G_{2b}'$ and we have, from the premisses of (Comp):

(i) $\Gamma \vdash_{\mathcal{T}_u} G_{2a} \xrightarrow{\Lambda_a} G_{2a}'$ for some Λ_a
(ii) $\Gamma \vdash_{\mathcal{T}_u} G_{2b} \xrightarrow{\Lambda_b} G_{2b}'$ for some Λ_b
(iii) $\texttt{Cond}_I(\Lambda.\texttt{sync}, \Lambda_a.\texttt{sync}, \Lambda_b.\texttt{sync}, G_2)$
(iv) $\texttt{Cond}(\Gamma, G_2)$
(v) $\texttt{Cond}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma, G_{2a}, G_{2b})$

Since $\texttt{Cond}_I(\Lambda, \Lambda_a, \Lambda_b, G_2')$, $\texttt{seval}(\Lambda_a \cup \Lambda_b) = \Lambda$.
Since $r \in G_2.\texttt{unbound}$, $r \in G_{2a}.\texttt{unbound}$ (exclusive) or $r \in G_{2b}.\texttt{unbound}$. Without loss of generality, suppose $r \in G_{2a}.\texttt{unbound}$, and therefore $\hat{r} : \texttt{rmv}\langle L\rangle \in \Lambda_a.\texttt{sync}$.
Let $G_{2a}' = N_{p_a}(G_{1a}')$; $G_{2b}' = N_{p_b}(G_{1b}')$; we can then rewrite G_2' as $G_2' = G_{2a}' \parallel G_{2b}' = N_{p'}(G_{1a}') \parallel N_{p_b}(G_{1b}')$.
From our induction hypothesis:

$$\Gamma \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_{p_a}(G_{1a}'')$$

with $G_{1a}' \equiv G_{1a}'' \parallel L$.
In addition, $\texttt{Cond}_I(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle, \langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle, \Lambda_b, G_2)$ holds (notice that the removed interaction could *not* be matched by Λ_b, since $r \in G_2.\texttt{unbound}$ and $r \in G_{2a}.\texttt{unbound}$, therefore $r \notin G_{2b}.\texttt{roles}$, and by hypothesis, $r \in G_2.\texttt{unbound}$). $\texttt{Cond}(\Gamma, G_2)$ and $\texttt{Cond}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma, G_{2a}, G_{2b})$ still hold.
Therefore (Comp) can apply, and

$$\Gamma \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_{p_a}(G_{1a}'') \parallel N_{p_b}(G_{1b}')$$

From Lemma 23,

$$\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G_{1a}'' \parallel G_{1b}')$$

Case (Ctx): In that case, $G_2 = G_{2a} \parallel G_{2b}$, $G_2' = G_{2a}' \parallel G_{2b}$ and we have, from the premisses of (Ctx):

(i) $\Gamma \vdash_{\mathcal{T}_u} G_{2a} \xrightarrow{\Lambda_a} G_{2a}'$ for some Λ_a
(ii) $\texttt{Ind}_I(\Lambda.\texttt{sync}, G_2)$
(iii) $\texttt{Cond}(\Gamma, G_2)$
(iv) $\texttt{Ind}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})$

Let $G_{2a}' = N_{p_a}(G_{1a}')$; $G_{2b} = N_{p_b}(G_{1b})$. Therefore, we can rewrite G_2' as $G_2' = G_{2a}' \parallel G_{2b} = N_{p_a}(G_{1a}') \parallel N_{p_b}(G_{1b}) = N_p(G_{1a}' \parallel G_{1b})$.
From our induction hypothesis:

$$\Gamma \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_{p_a}(G_{1a}'')$$

with $G_{1a}' \equiv G_{1a}'' \parallel L$, therefore $G_{2'} = N_p(G_{1a}'' \parallel L \parallel G_{1b})$.
In addition, $\texttt{Ind}_I(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \cup \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle.\texttt{sync}, G_2)$ holds (by hypothesis, $r \in G_2.\texttt{unbound}$). Also, $\texttt{Cond}(\Gamma, G_2)$ holds after (iii) and

$$\begin{aligned}&\texttt{Ind}_P(, \langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle.\texttt{prior},\\ &\quad \langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r} : \texttt{rmv}\langle n(L)\rangle\}\rangle.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})\\ \Leftrightarrow &\texttt{Ind}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})\end{aligned}$$

which holds, from (iv) above.

Therefore (Ctx) can apply, and

$$\Gamma \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_{p_a}(G''_{1a}) \parallel N_{p_b}(G_{1b})$$

From Lemma 23,

$$\Gamma \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\texttt{rmv}\langle n(L)\rangle\}\rangle} N_p(G''_{1a} \parallel G_{1b})$$

□

Lemma 26. *For any Γ_2, any second order graph $G_2 = N_p(G_1)$ and $G'_2 = N_p(G'_1)$ with $L \in \mathbb{L}$, $L \notin G_2$, Λ; if $\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle \in \Lambda.\texttt{sync}$ and $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} G'_2$, then, either*

(i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(G'_1 \parallel L)$*; or*
(ii) $G_2 = N_p(G'_1 \parallel L)$ *and* $\Lambda = \langle \emptyset, \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle$.

Proof. By induction on the reduction rule of $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} G'_2$.

Case (Trans): In that case, G_2 is a single location, therefore $G_2 = N_p(G_1) = n(G_1)$.

From the premisses of (Trans):

(i) $\Gamma.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} N_p(G'_1) \in \mathcal{T}_2$
(ii) $\Sigma(G_2) \in \Gamma_2.\texttt{graph}$
(iii) $\texttt{Auth}_2(\Gamma_2.\texttt{graph}, \Gamma_2.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\Lambda} N_p(G'_1 \parallel L))$

By definition of $\mathcal{T}_2$, $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} H_1$ and $L \in H_1$ (for a given H_1) and $L \notin G'_1$. In that case, the rule $\Gamma_2.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(G'_1 \parallel L)$ is also in $\mathcal{T}_2$.

An other possibility is that $\Gamma_2.\texttt{names} \cdot \emptyset \rhd n(G_1) \xrightarrow{\Lambda} n(H_1)$ with $H_1 \subsetneq G_1$, and, for each $L_i \in G_1$ such that $L_i \notin H_1$, $\hat{r_i} : \overline{\texttt{rmv}}\langle n(L)\rangle \in \Lambda.\texttt{sync}$. In that case, if L is the only location that is not in H_1, then $G_1 = H_1 \parallel L$, and the proof is finished here, as item (ii) holds. Otherwise, $G_1 = H_r \parallel L \parallel H_1$, with $H_r \neq \emptyset$ containing all other locations that are removed. Therefore, $\Gamma_2.\texttt{names} \cdot \emptyset \rhd G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(H_1 \parallel L)$ is also in $\mathcal{T}_2$.

In any cases, the authorisation function still holds[a]. Therefore,

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \cup \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(G'_1)$$

Case (Comp): In that case, $G_2 = G_{2a} \parallel G_{2b}$, $G'_2 = G'_{2a} \parallel G'_{2b}$ and we have, from the premisses of (Comp):

(i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda_a} G'_{2a}$ for some Λ_a
(ii) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2b} \xrightarrow{\Lambda_b} G'_{2b}$ for some Λ_b
(iii) $\texttt{Cond}_I(\Lambda.\texttt{sync}, \Lambda_a.\texttt{sync}, \Lambda_b.\texttt{sync}, G_2)$
(iv) $\texttt{Cond}(\Gamma_2, G_2)$
(v) $\texttt{Cond}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})$

Since $\texttt{Cond}_I(\Lambda, \Lambda_a, \Lambda_b, G'_2)$, $\texttt{seval}(\Lambda_a \cup \Lambda_b) = \Lambda$.

Since $r \in G_2.\texttt{unbound}$, $r \in G_{2a}.\texttt{unbound}$ (exclusive) or $r \in G_{2b}.\texttt{unbound}$. Without loss of generality, suppose $r \in G_{2a}.\texttt{unbound}$ and $G'_{2a} = N_{p_a}(G'_{1a})$.

Let $G'_{2b} = N_{p_b}(G'_{1b})$; we can rewrite G'_2 as $G'_2 = G'_{2a} \parallel G'_{2b} = N_{p_a}(G'_{1a}) \parallel N_{p_b}(G'_{1b})$.

From our induction hypothesis, either:

(i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_{p_a}(G'_{1a} \parallel L)$; or
(ii) $G_{2a} = N_{p_a}(G'_{1a} \parallel L)$.

In the first case: $\texttt{Cond}_I(\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle, \langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{r : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle, \Lambda_b, G_2)$ holds (notice that the removed interaction could *not* be matched by Λ_b, since it appears in Λ; and by hypothesis, $r \in G_2.\texttt{unbound}$). $\texttt{Cond}(\Gamma_2, G_2)$ and $\texttt{Cond}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})$ still hold.

Therefore (Comp) can apply, and

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \cup \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_{p_a}(G'_{1a} \parallel L) \parallel N_{p_b}(G'_{1b})$$

From Lemma 23,

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \cup \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(G'_{1a} \parallel L \parallel G'_{1b}) = N_p(G'_1 \parallel L)$$

which ends the proof.
If $G_{2a} = N_{p_a}(G'_{1a} \parallel L)$, then $\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle = \langle \emptyset, \emptyset\rangle$. In that case, we can apply (Ctx): $\texttt{Ind}_P(, \Lambda.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma_2, G_b, G_a)$ holds, since $\texttt{Cond}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Lambda_b.\texttt{prior}, \Gamma_2, G_a, G_b)$ holds. $\texttt{Ind}_I(\Lambda_b.\texttt{sync}, G_{2a} \parallel G_{2b})$ holds since $\texttt{Cond}_I(\Lambda, \Lambda_a, \Lambda_b, G_{2a} \parallel G_{2b})$ holds and $\Lambda_a = \langle \emptyset, \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle$. $\texttt{Cond}(\Gamma_2, G_{2a} \parallel G_{2b})$ still holds.
We then have

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_{p_a}(G'_{1a} \parallel L) \parallel G'_{1b}$$

with $\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle = \Lambda_b$

Case (Ctx): In that case, $G_2 = G_{2a} \parallel G_{2b}$, $G'_2 = G'_{2a} \parallel G_{2b}$ and we have, from the premisses of (Ctx):

(i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda_a} G'_{2a}$ for some Λ_a such that $\Lambda_a.\texttt{sync} = \Lambda.\texttt{sync}$
(ii) $\texttt{Ind}_I(\Lambda.\texttt{sync}, G_2)$
(iii) $\texttt{Cond}(\Gamma_2, G_2)$
(iv) $\texttt{Ind}_P(, \Lambda.\texttt{prior}, \Lambda_a.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})$

Let $G'_{2a} = N_{p_a}(G'_{1a})$.
From our induction hypothesis, either:

(i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_{p_a}(G'_{1a} \parallel L)$; or
(ii) $G_2 = N_{p_a}(G'_{1a} \parallel L)$.

In the first case, $\texttt{Ind}_I(\langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \setminus \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle.\texttt{sync}, G_2)$ holds (by hypothesis, $r \in G_2.\texttt{unbound}$). $\texttt{Cond}(\Gamma_2, G_2)$ and $\texttt{Ind}_P(, \langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle.\texttt{prior}, \langle \Lambda_a.\texttt{prior}, \Lambda_a.\texttt{sync} \cup \{\hat{r} : \overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle.\texttt{prior}, \Gamma_2, G_{2a}, G_{2b})$ still hold.
Therefore (Ctx) can apply, and

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \setminus \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_{p_a}(G'_{1a} \parallel L) \parallel N_{p_b}(G_{1b})$$

From Lemma 23,

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\langle \Lambda.\texttt{prior}, \Lambda.\texttt{sync} \cup \{\hat{r}:\overline{\texttt{rmv}}\langle n(L)\rangle\}\rangle} N_p(G'_{1a} \parallel L \parallel G_{1b}) = N_p(G'_1 \parallel L)$$

In the second case, we have $G_{2a} = N_{p_a}(G'_{1a} \parallel L)$, therefore $G_2 = N_{p_a}(G'_{1a} \parallel L) \parallel N_{p_b}(G_{1b}) = N_p(G'_1 \parallel L)$, which ends the proof.

□

[a] The argument is the same than case (Trans) in the proof of Lemma 25.

Theorem 3 (The 1st order graph simulates the 2nd order graph)**.** *$\mathcal{S}^{-1}$ is a partial $\langle \rightleftharpoons_E^{-1}, \rightleftharpoons_\lambda^{-1}\rangle$-simulation, with respect to $\{\Lambda | \texttt{prune}(\Lambda) = \Lambda\}$.*

Proof. We have to show the five items of Definition 46.
We show directly item (i). Let $\langle G_2, G_1\rangle \in \mathcal{S}^{-1}$. By definition of $\mathcal{S}^{-1}$, $G_2 = N_p(G_1)$.
We are given a 2nd order environment $\Gamma_2 = \Delta_2 \cdot G_2^s$. We have to show that there exists $\Gamma_1 = \Delta_1 \cdot G_1^s$.
Since $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G'_2$, $\Sigma(G_2) \in G_2^s$. Since $\langle G_2, G_2\rangle \in \mathcal{S}^{-1}$, $G_2 = N_p(G_1)$, therefore $\Sigma(N_p(G_1)) \subseteq G_2^s$. Therefore, $e(\Sigma(N_p(G_1))) = \Sigma(G_1) \subseteq e(G_2^s)$, and, by definition, $G_2^s \rightleftharpoons_s^{-1} e(G_2^s)$.
Concerning Δ_1, if $\Delta_1 = \Delta_2 \cap \mathbb{R}$, then $\texttt{supp}(G_1) = \texttt{supp}(G_2) \cap \mathbb{R} \subseteq \Delta_2 \cap \mathbb{R}$.
Item (ii) is trivial, since $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} G'_2$, and $\Lambda_2 = \Lambda_2$, then $\Lambda_2 \rightleftharpoons_\lambda \texttt{prune}(\Lambda_2)$.
Therefore, we still have to show that, for all Γ_1 such that $\Gamma_2 \rightleftharpoons_E^{-1} \Gamma_1$, $\texttt{supp}(G_1) \subseteq \Gamma_1.\texttt{names}$ and $\Sigma(G_1) \subseteq \Gamma_1.\texttt{graph}$, $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_2} G'_1$ (item (iii)), with $G'_2 \rightleftharpoons_E^{-1} G'_1$ (item (iv)). From Lemma 24, item (iv) implies item (v); we therefore only prove item (iv).
We prove those two items by induction on the depth of G_2[a]:

Base case (depth 1): We have $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} C_2$ and G_2 is a single location, so the rule (Trans) applies. Hence, from the premisses of the rule, we have $\Gamma_2 \rhd G_2 \xrightarrow{\Lambda_2} C_2$ and $\texttt{Auth}_2(\Gamma_2, \Sigma(G_2), \Lambda_2, C_2)$ holds.

Since $\Gamma_2 \rhd G_2 \xrightarrow{\Lambda_2} C_2$ and $\Lambda_2 = \texttt{prune}(\Lambda_2)$, after Lemma 20, then there exists G_1, C_1, Γ_1 and Λ_1 such that $G_2 = n(G_1) = N_p(G_1)$, $C_2 = N_p(C_1)$, $\Lambda_1 = \Lambda_2$ and $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_1} C_1$.

Inductive case $(n+1)$:

Induction Hypothesis (IH1). For all G_2 with a depth of at most n, for all Γ_2, G_2' and Λ such that $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} G_2'$, for $G_1 = N_p^{-1}(G_2)$, for all Γ_1 such that $\Gamma_1 =_E \Gamma_2$, $\texttt{supp}(G_1) \subseteq \Gamma_1.\texttt{names}$ and $\Sigma(G_1) \subseteq \Gamma_1.\texttt{graph}$, there exists G_1' such that $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda} G_1'$. ▷

We have $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\Lambda_2} C_2$. Two rules can apply, (Comp) or (Ctx):

Case (Comp): In this case, we have $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\Lambda_2} C_{2a} \parallel C_{2b}$ with $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda_{2a}} C_{2a}$, $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2b} \xrightarrow{\Lambda_{2b}} C_{2b}$, and such that $\Lambda_2 = \texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b}) = \texttt{prune}(\Lambda_2)^b$.

By hypothesis, $C_{2a} = N_{p_a}(C_{1a})$ and $C_{2b} = N_{p_b}(C_{1b})$ such that $C_1 = C_{1a} \parallel C_{1b}$.

Since $\texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b}) = \texttt{prune}(\Lambda_2)$, then any action with role in $\mathbb{R}_2$ in Λ_{2a} is matched in Λ_{2b} (and vice-versa). We show items (iii) and (iv) by induction on the number n_a of action with role in $\mathbb{R}_2$ in Λ_{2a} (and also in Λ_{2b}, since all are matched).

Base case ($n_a = 0$): Since $n_a = 0$, then $\texttt{prune}(\Lambda_{2a}) = \Lambda_{2a}$ and $\texttt{prune}(\Lambda_{2b}) = \Lambda_{2b}$.

From the induction hypothesis (IH1), for all G_{1a} such that $\langle G_{2a}, G_{1a}\rangle \in \mathcal{S}^{-1}$, $\Gamma_1 \vdash_{\mathcal{T}_1} G_{1a} \xrightarrow{\Lambda_{2a}} C_{1a}$ such that $\langle C_{2a}, C_{1a}\rangle \in \mathcal{S}^{-1}$, and similarly for G_{2b}.

Since $\Lambda_2.\texttt{sync.roles} \subset \mathbb{R}$, $G_1.\texttt{unbound} = G_2.\texttt{unbound} \cap \mathbb{R}$, and $\Lambda_2.\texttt{sync.roles} \subseteq G_2.\texttt{unbound}$, then $\Lambda_2.\texttt{sync.roles} \subset G_1.\texttt{unbound}$. And, as stated above, $\Lambda = \texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b})$. Therefore $\texttt{Cond}_I(\Lambda_2, \Lambda_{2a}, \Lambda_{2b}, G_1)$ holds.

By hypothesis, $\texttt{Cond}(\Gamma, G_1)$ holds.

Therefore, $\Gamma_1 \vdash_{\mathcal{T}_1} G_{1a} \parallel G_{1b} \xrightarrow{\Lambda_2} C_{1a} \parallel C_{1b}$, which proves item (iii).

In addition, $C_2 = C_{2a} \parallel C_{2b} = N_p(C_{1a}) \parallel N_p(C_{1b}) = N_{p'}(C_1)$. Therefore $\langle C_2, C_1\rangle \in \mathcal{S}^{-1}$ (item (iv)).

Inductive case ($n_a > 0$):

Induction Hypothesis (IH2). For all Λ_{IHa}, Λ_{IHb} such that Λ_{IHa} and Λ_{IHb} contain at most n_a actions with role in $\mathbb{R}_2$, and such that $\texttt{seval}(\Lambda_{IHa} \cup \Lambda_{IHb}) = \Lambda = \texttt{prune}(\Lambda)$; if $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda_{IHa}} N_{p_a}(C_{1a})$ and $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2b} \xrightarrow{\Lambda_{IHb}} N_{p_b}(C_{1b})$ then $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda_{IH}} C_{1a} \parallel C_{1b}$ and $\langle C_{1a} \parallel C_{1b}, N_{p_a}(C_{1a}) \parallel N_{p_b}(C_{1b})\rangle \in \mathcal{S}$. ▷

Our goal is to show that the above statement also holds if Λ_{2a} and Λ_{2b} have $n_a + 1$ actions with roles in $\mathbb{R}_2$. Intuitively, in that case, we show that we can remove an action a of Λ_{2a} and its conjugate $\overline{a}$ from Λ_{2b}. Without loss of generality, suppose $a = r : \texttt{rmv}\langle n(L)\rangle$ and $\overline{a} = \overline{r} : \overline{\texttt{rmv}}\langle n(L)\rangle$.

Let $\Lambda_{2a}^- = \langle \Lambda_{2a}.\texttt{prior}, \Lambda_{2a}.\texttt{sync} \setminus \{a\}\rangle$ (resp. for Λ_{2b}^- and $\overline{a}$).

We have $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda_{2a}} N_{p_a}(C_{1a})$. From Lemma 25, $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\langle \Lambda_{2a}.\texttt{prior}, \Lambda_{2a}.\texttt{sync} \setminus \{a\}\rangle} N_{p_a}(C_{1a}')$ with $C_{1a} \equiv C_{1a}' \parallel L$.

Similarly for G_{2b}, we have $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2b} \xrightarrow{\Lambda_{2b}} N_{p_b}(C_{1b})$. Therefore, from Lemma 26, either (i) $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2b} \xrightarrow{\langle \Lambda_{2b}.\texttt{prior}, \Lambda_{2b}.\texttt{sync} \setminus \{\overline{a}\}\rangle} N_{p_b}(C_{1b}')$ with $C_{1b}' \equiv C_{1b} \parallel L$; or (ii) $G_{2b} = N_{p_b}(C_{1b} \parallel L)$.

Case (i): Notice that $\texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b}) = \texttt{seval}(\langle \Lambda_{2a}.\texttt{prior}, \Lambda_{2a}.\texttt{sync} \setminus \{a\}\rangle \cup \langle \Lambda_{2b}.\texttt{prior}, \Lambda_{2b}.\texttt{sync} \setminus \{\overline{a}\}\rangle)$.

Therefore, if

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b})} N_{p_a}(C_{1a}' \parallel L) \parallel N_{p_b}(C_{1b})$$

then, using (Comp)

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\texttt{seval}(\Lambda_{2a}^- \cup \Lambda_{2b}^-)} N_{p_a}(C_{1a}') \parallel N_{p_b}(C_{1b} \parallel L)$$

We have that $C_2 = N_{p_a}(C_{1a}' \parallel L) \parallel N_{p_b}(C_{1b}) = N_p(C_{1a}' \parallel C_{1b} \parallel L) = N_p(C_1)$, and $N_{p_a}(C_{1a}') \parallel N_{p_b}(C_{1b} \parallel L) = N_{p'}(C_{1a}' \parallel C_{1b} \parallel L) = N_{p'}(C_1)$ for a $p' \neq p$

Therefore, from (IH2), $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda=\texttt{seval}(\Lambda_{2a}^- \cup \Lambda_{2b}^-)} C'_{1a} \parallel L \parallel C_{1b} = C_1$ and $\langle C_1, N_{p'}(C_1)\rangle \in \mathcal{S}$.

Therefore, $\Gamma_1 \vdash_{\mathcal{T}_1} G_1 \xrightarrow{\Lambda=\texttt{seval}(\Lambda_{2a} \cup \Lambda_{2b})} C'_{1a} \parallel L \parallel C_{1b} = C_1$ and $\langle C_1, N_p(C_1)\rangle \in \mathcal{S}$.

Case (ii): In that case, a (resp. $\overline{a}$) is the only interaction with its role in $\mathbb{R}_2$ in Λ_{2a} (resp. Λ_{2b}). Therefore Λ_{2a}^- contains no interaction with its role in $\mathbb{R}_2$.
Rule (Ctx) applies, and

$$\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\Lambda} N_{p_a}(C'_{1a}) \parallel N_{p_b}(C_{1b} \parallel L)$$

Notice that, here, we can not apply our (outer) induction hypothesis and conclude, since the depth of the reduction tree is the same.
However, we can apply the case for rule (Ctx) below.

Case (Ctx): By symmetry, consider that G_{2a} reduces and G_{2b} remains the same. We have $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \parallel G_{2b} \xrightarrow{\Lambda} C_{2a} \parallel G_{2b}$, with $G_{2a} \parallel G_{2b} = G_2 = N_p(G_1)$. We have that $G_{2a} = N_{p_a}(G_{1a})$ and $G_{2b} = N_{p_b}(G_{1b})$ for some partition function p'.
From the premisses of (Ctx), we know that $\Gamma_2 \vdash_{\mathcal{T}_2} G_{2a} \xrightarrow{\Lambda} C_{2a}$. From the induction hypothesis (IH1), we know that $\Gamma_1 \vdash_{\mathcal{T}_1} G_{1a} \xrightarrow{\Lambda} C_{1a}$ with $C_{2a} = N_{p_a}(C_{1a})$.
Thus, using rule (Ctx) with the first order graph: $\Gamma_1 \vdash_{\mathcal{T}_1} G_{1a} \parallel G_{1b} \xrightarrow{\Lambda} C_{1a} \parallel G_{1b}$.
Finally, using Corollary 3, $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda} N_{p_a}(C_{1a}) \parallel N_{p_b}(G_{1b})$ if and only if $\Gamma_2 \vdash_{\mathcal{T}_2} G_2 \xrightarrow{\Lambda_2} N_p(C_{1a} \parallel G_{1b})$.

□

[a] Notice that it is not possible to do a proof by case analysis of the reduction rule used, as we need to use the cases recursively.

[b] Notice that, here, we have no guarantee that Λ_{2a} and Λ_{2b} are pruned (typically, a location can be exchanged from G_{2a} to G_{2b}, for instance), therefore, we can not apply the induction hypothesis on G_{2a} and G_{2b}. Our goal here is first to show that there is, however, a similar reduction without exchange, on which we could apply the induction hypothesis.

Theorem 4 (Partial bisimulation)**.** *Let $p \in \mathbb{P}_c^{\mathbb{G}}$. Let $N_p(G)$ be equipped with the semantics of 2nd order graph. Let $\mathcal{R} = \{\langle G, N_p(G)\rangle | G \in \mathbb{G}\}$.*
$\mathcal{R}$ is a partial bisimulation with respect to $\{\Lambda | \Lambda \in \Lambda_2 \wedge \texttt{prune}(\Lambda) = \Lambda\}$.

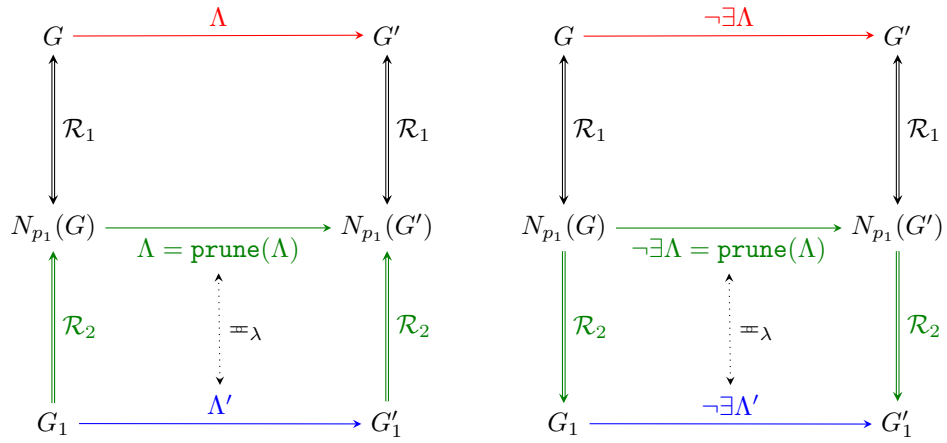
Proof. The proof follows directly from Theorem 2 and Theorem 3. □

Interpretation of the bisimulation result. We claim the partial bisimulation result above is a fundamental result in our work. In the context of isolation, it can be used in various ways: analysing the isolation property of a system, having different *views* of the same system, etc.. This paragraph highlight these various approaches.

As a preliminary, we should temper the content below, as the partial bisimulation result gives us a relation between a location graph and its nested counterpart. The second order graph may not be easy to work with. We suggest that the workflow should be the following: given a location graph G, take any partition function p based on our intuition; use our result to support that G and $N_p(G)$ are partially bisimilar; find a location graph G_2 that highlights an interesting execution, and show that $N_p(G)$ simulates G_2; deduce that G can take this execution. Conversely, if we are interested in showing an execution is *not* possible, we have to take a G_2 in which this transition is not possible, and show that G_2 simulates $N_p(G)$. Figure 3.7 illustrate the two strategies explained here.

This workflow is not proofless: we have to prove $N_p(G)$ simulates G_2. Hopefully, this proof would be easy in most cases (taking a subset of $N_p(G)$ transitions, etc.).

First, the bisimulation between a graph and it nested counterpart can be used to analyse isolation policies. Consider we are given a system S represented as a location graph G_S, for which we would like to analyse the isolation policy. Usually, we have a good intuition of which locations should be allowed to interact and which should not. That is, we have an idea of the partition function, and the objective is to assert that no unallowed interaction occurs. In order to analyse the isolation property, one could abstract



(a) Workflow to show the existence of a transition.

(b) Workflow to show the absence of a transition.

Figure 3.7: Illustration of the workflow to work with nested graphs. In Subfigure 3.7a, we are interested in showing the existence of the red transition. Given a location graph G, if we prove that $N_{p_1}(G)$ simulates G_1 (the green arrows), then for any transition of G_1 (the blue arrow), the original graph G can take a similar transition. In Subfigure 3.7b, we are interested in showing the absence of of the red transition. Given a location graph G, if we prove that G_1 simulates $N_{p_1}(G)$ (the green arrows), then if there is no transitions in G_1 with a label equivalent to Λ, then G can not reduce with Λ.

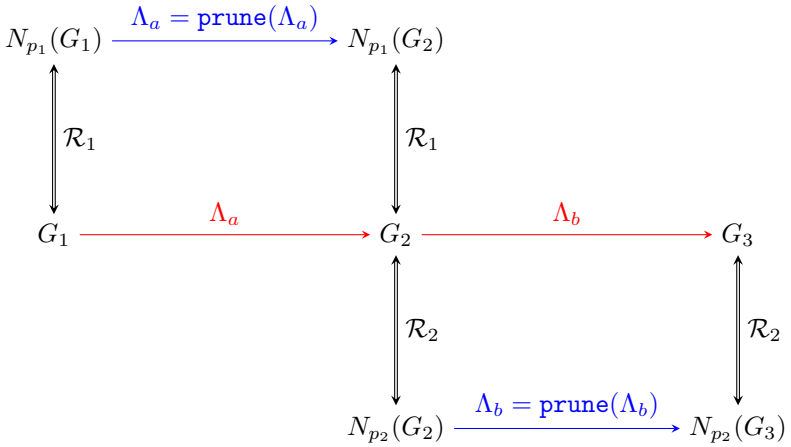


Figure 3.8: Illustration of using different views of the same system. Given the blue arrows, the partial simulation shows that the original system can take the transitions: we show the existence of the red arrows.

away details, which is done by *nesting* G_S to obtain a new location graph G'_S. Our partial bisimulation result means that G'_S also represent S. An example of such approach is shown in the following chapter, in Section 4.1.1. In this section, we analyse the actor model example, where we aggregate all owned objects into a nested location, and we show this aggregate location only communicates with its owner.

A second way the bisimulation can be used is to have multiple different views of a single system. Consider that we want to reason about multiple different ways to encapsulate a single system S (represented as a location graph G_S). An example of such a reasoning happens when a system implements a policy (e.g. owner-as-dominator) with a monitoring device (which is allowed to overcome the policy). In such cases, there are multiple possible partition functions (e.g. p_1 and p_2). Our result highlights that these multiple views (e.g. $N_{p_1}(G_S)$ and $N_{p_2}(G_S)$) are equivalent. In addition, it provides evidence that we can freely switch between views (e.g. consider a partial execution $N_{p_1}(G_S)$ to $N_{p_1}(G'_S)$, then $N_{p_2}(G'_S)$ to $N_{p_2}(G''_S)$). Figure 3.8 shows an example of this approach.

3.3.4 Multiple levels of nesting

So far, we discussed how to create a single level of nesting. To conclude this section, we propose ideas to nest a graph on multiple layers. Of course, being general, our method could be applied to 2^{nd} order graphs to create 3^{rd} order graph, etc.. Unfortunately, our method does not provide any mechanism to change the layer of a location (e.g. so that a 2^{nd} order location becomes a 3^{rd} order location). Even

worse: by essence, the 3rd order graph would not even know that it is a 3rd order graph; for instance, administrative roles of one layer are not the same than those of the other layers.

In this section, we first explain a bit more precisely the intuition developed above. Secondly, we propose, informally, a more suitable mechanism for higher order nesting. The intuition for this mechanism is to use a common set of administrative role across layers, and let one layer of nesting intercept administrative interactions of an other layer.

The failure of naive recursive nesting. It is possible to extend the definitions of location nesting function (Definition 49) and graph nesting function (Definition 55) to have multiple levels of nesting. The extended definitions are mutually recursive.

Definition 67 (Recursive location nesting function)**.**

$$\Pi_p(G) = \begin{cases} n(\mathbb{N}_p(G)) & \text{if } p(G) \neq \{G\} \\ n(G) & \text{otherwise} \end{cases}$$

Definition 68 (Recursive graph nesting function)**.** $\mathbb{N}_p(G) = \prod_{G_i \in p(G)} \Pi_p(G_i)$

Remark. The two definitions 67 and 68 are conservative extensions of Definitions 49 and 55: if we have p such that, $\forall G_i \in p(G), p(G_i) = G_i$, we have $\Pi_p(G_i) = n(G_i)$, hence we fall back on $\mathbb{N}_p(G) = \prod_{G_i \in p(G)} n(G_i) = N_p(G)$. ◁

Remember Table 3.1 on page 46, we have that roles of the 2nd order graphs are taken in $\mathbb{R} \cup \mathbb{R}_2$, where roles in $\mathbb{R}_2$ are used for out-of-band communications (communications that are added for the simulation, such as the removal of a part of the nested graph, in order to perform change of part of the graph).

Now, if we have $n(n(A) \parallel n(B))$ with $\mathbb{R}_A$ the set of roles of A and $\mathbb{R}_B$ those of B. Thus, the set of roles of $n(A)$ is $\mathbb{R}_A \cup \mathbb{R}_{2A}$ (for a suitable $\mathbb{R}_{2A}$), and $\mathbb{R}_B \cup \mathbb{R}_{2B}$ for $n(B)$. With the same reasoning, the set of roles of $n(n(A) \parallel n(B))$ is $\mathbb{R}_A \cup \mathbb{R}_{2A} \cup \mathbb{R}_B \cup \mathbb{R}_{2B} \cup \mathbb{R}'$.

For this reason, we can not change the level of nesting of a given location. Therefore, this simple idea is not adequate.

One could be tempted to define higher-order semantics such as how we defined 2nd order semantics. The set of unconstrained location transitions (called $\mathcal{T}_{\text{HO}}$) would be defined as follow:

$$\mathcal{T}_{\text{HO}} = \{\Gamma_2 \rhd \Pi_p(G) \xrightarrow{\Lambda} \mathbb{N}_p(H) | \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G'\} \cup \{\Gamma \rhd \Pi_p(G) \xrightarrow{\langle \varepsilon, \hat{r}:\overline{\texttt{rmv}}\langle \Pi_p(G) \rangle\rangle} \emptyset\}$$

Such an approach is, however, very different from the one taken, even tho the difference is subtle and could be unnoticed at first glance. Consider a 1st order graph $A \parallel B \parallel C$ such that $\Gamma \vdash_{\mathcal{T}_u} A \xrightarrow{\Lambda_A} A'$ and $\Gamma \vdash_{\mathcal{T}_u} B \xrightarrow{\Lambda_B} B'$ (we do not need to detail further the environment, the labels and the locations). We hence have $\Gamma \vdash_{\mathcal{T}_u} A \parallel B \parallel C \xrightarrow{\Lambda} A' \parallel B' \parallel C$. Finally, let p be a partition function such that $p(A \parallel B \parallel C) = \{A, B \parallel C\}$ and $p(B \parallel C) = \{B, C\}$. Therefore, we have $\mathbb{N}_p(A \parallel B \parallel C) = \Pi_p(A) \parallel \Pi_p(B \parallel C) = n(A) \parallel n(\mathbb{N}_p(B \parallel C))$.
The derivation tree of the higher-order graph is then:

$$\text{(Comp)}\ \frac{\text{(Trans)}\ \dfrac{\text{Def of } \mathcal{T}_{\text{HO}}\ \dfrac{\text{Hypothesis}\ \overline{\qquad}}{\Gamma \vdash_{\mathcal{T}_u} A \xrightarrow{\Lambda_A} A'}}{\Gamma' \vdash_{\mathcal{T}_{\text{HO}}} \Pi_p(A) \xrightarrow{\Lambda'_A} \mathbb{N}_p(A')} \qquad \text{(Trans)}\ \dfrac{\text{Def of } \mathcal{T}_{\text{HO}}\ \dfrac{\text{Hypothesis}\ \overline{\qquad}}{\Gamma \vdash_{\mathcal{T}_u} B \parallel C \xrightarrow{\Lambda_B} B' \parallel C}}{\Gamma' \vdash_{\mathcal{T}_{\text{HO}}} \Pi_p(B \parallel C) \xrightarrow{\Lambda'_B} \mathbb{N}_p(B' \parallel C)}}{\Gamma' \vdash_{\mathcal{T}_{\text{HO}}} \Pi_p(A) \parallel \Pi_p(B \parallel C) \xrightarrow{\Lambda'} \Pi_p(A') \parallel \mathbb{N}_p(B' \parallel C)}$$

If we only use the 2nd order semantics, we have $\Pi_p(B \parallel C) = n(\mathbb{N}_p(B \parallel C)) = n(n(B) \parallel n(C))$, and

the derivation tree is:

$$\dfrac{\dfrac{\textsc{Hypothesis}\ \dfrac{}{\Gamma \vdash_{\mathcal{T}_u} A \xrightarrow{\Lambda_A} A'}}{\textsc{Def of } \mathcal{T}_u\ \dfrac{}{\Gamma' \rhd n(A) \xrightarrow{\Lambda'_A} n(A')}}}{\textsc{(Trans)}\ \dfrac{}{\Gamma' \vdash_{\mathcal{T}_u} n(A) \xrightarrow{\Lambda'_A} n(A')}} \quad \dfrac{\dfrac{\textsc{Hypothesis}\ \dfrac{}{\Gamma \vdash_{\mathcal{T}_u} B \xrightarrow{\Lambda_B} B'}}{\textsc{Def of } \mathcal{T}_u\ \dfrac{}{\Gamma' \rhd n(B) \xrightarrow{\Lambda'_B} n(B')}}}{\textsc{(Trans)}\ \dfrac{}{\Gamma' \vdash_{\mathcal{T}_u} n(B) \xrightarrow{\Lambda'_B} n(B')}}}{\textsc{(Ctx)}\ \dfrac{}{\Gamma' \vdash_{\mathcal{T}_u} n(B) \parallel n(C) \xrightarrow{\Lambda'_B} n(B') \parallel n(C)}}}{\textsc{Def of } \mathcal{T}_u\ \dfrac{}{\Gamma'' \rhd n(n(B) \parallel n(C)) \xrightarrow{\Lambda''_B} n(n(B') \parallel n(C))}}}{\textsc{(Trans)}\ \dfrac{}{\Gamma'' \vdash_{\mathcal{T}_u} n(n(B) \parallel n(C)) \xrightarrow{\Lambda''_B} n(n(B') \parallel n(C))}}}{\textsc{(Comp)}\ \dfrac{}{\Gamma'' \vdash_{\mathcal{T}_{\text{HO}}} \mathbb{n}_p(A) \parallel \mathbb{n}_p(B \parallel C) \xrightarrow{\Lambda'} \mathbb{n}_p(A') \parallel \mathbb{N}_p(B' \parallel C)}}$$

Two points motivate that we do not use such semantics in the work:

— It is not necessary. As shown in the example above, the 2nd order semantics remains usable even with higher-order graphs.

— From an intuitive point of view, $\mathcal{T}_{\text{HO}}$ changes the way to think about nesting locations: it allows to inspect across multiple levels of nesting. If $n(A)$ reduces to $n(B)$, it is not because A reduces to B anymore.

Nonetheless, the two semantics being equivalent would not be a surprise[a]. The difference lies more on how to think of higher-level graphs than on the real differences of expressivity.

[a] The proof is probably short, but not formally done.

Higher-order nesting. This last paragraph discuss some directions for future work for higher order nesting. The very nature of such discussion implies that expectations developed here are not proven. Therefore, propositions and conjectures introduced here should be taken as educated guesses, not results.

In the previous paragraph, we showed that the problem with a naive recursive nesting is that administrative roles of one level is considered a regular role in the next level. An intuitive solution would be to allow one layer to capture and analyse administrative messages of the graph it nests, and to possibly intercept those messages.

We slightly adapt the intuition behind the nesting functions $\mathbb{n}_p(G)$ and $\mathbb{N}_p(G)$ defined in Definitions 67 and 68, into a new nesting function $\nu_p(G)$ to take into account the fact that administrative roles should be the same across nesting levels.

Definition 69 (Higher-order nesting function)**.**

$$\nu_p(G) = \begin{cases} \prod_{G_i \in p(G)} [\nu_p(G_i) : \Sigma(G_i) \lhd p_i \bullet r_i] & \text{if } p(G) \neq \{G\} \\ [G_i : \Sigma(G_i) \lhd p_i \bullet r_i] & \text{otherwise} \end{cases}$$

where p_i (resp. r_i) is such that $p_i \backslash G_i.\texttt{punbound} \subset \mathbb{R}_2$ (resp. $r_i \backslash G_i.\texttt{runbound} \subset \mathbb{R}_2$), and $p_i \cap r_i = \emptyset$.

We propose the following the unconstrained location transitions for a location $[G : \Sigma(G) \lhd p \bullet r] : \Delta \cdot \emptyset \rhd [G : G_s \lhd p \bullet r] \xrightarrow{\langle \pi, \iota \cup \iota'_2 \rangle} \nu_p(H)$ if $\Gamma \vdash_{\mathcal{T}_\star} G \xrightarrow{\langle \pi, \iota \cup \iota_2 \rangle} G'$, where ι contains actions on first-order roles, and ι_2 and ι'_2 contains actions on administrative roles, such that the four following propositions hold:

(i) $\forall \hat{r} : \overline{\texttt{rmv}}\langle L \rangle \in \iota_2 \cdot \hat{r} : \overline{\texttt{rmv}}\langle L \rangle \in \iota'_2 \Rightarrow L \in H$

(ii) $\forall \hat{r} : \texttt{rmv}\langle L \rangle \in \iota_2 \cdot \hat{r} : \texttt{rmv}\langle L \rangle \in \iota'_2$

(iii) $\forall \hat{r} : \overline{\texttt{rmv}}\langle L \rangle \in \iota'_2 \cdot \hat{r} : \overline{\texttt{rmv}}\langle L \rangle \in \iota_2$

(iv) $\forall \hat{r} : \texttt{rmv}\langle L \rangle \in \iota'_2 \cdot \hat{r} : \texttt{rmv}\langle L \rangle \in \iota'_2 \Rightarrow L \in H$

Remark. In the rule above, whether G is a first order subgraph or an higher-order graph is irrelevant, which is what we intend to do. However, the semantics of first-order is not the same as those of higher-order ($\mathcal{T}_1$ or $\mathcal{T}_H$). In this informal introduction, we note $\mathcal{T}_\star$ to highlight that this does not really matter.

Formally, we would probably devise two rules, one if the nested graph takes a $\mathcal{T}_1$, and the other is the nested graph takes a $\mathcal{T}_H$ transition. Since, in this section, we focus only on informal presentation, and we have nothing but intuition to support our claims, we leave actual details for future work. ◁

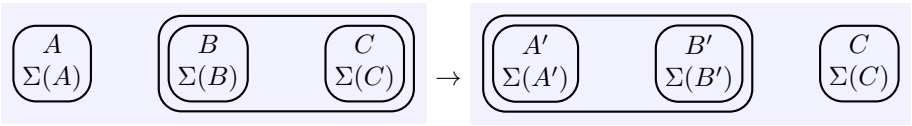


Figure 3.9: Example of a transition in which locations have to change their levels of nesting. In this example, A reduces to A' and B to B'. The overall starting graph is partitionned with A on one side, and B and C on the other side, while the resulting graph has A' and B' on one side, and C on the other.

The main difference with the nesting presented previously is that the nested graph takes a transition with some interactions on $\mathbb{R}_2$ (i.e. some administrative interactions), in ι_2. The outer location can add and remove some of those administrative interactions, and compensate properly the modifications. For instance, the condition (i) of the list above states that if the nested graph G gets rid of a location L (shown as the emission of the $\hat{r} : \overline{\texttt{rmv}}\langle L \rangle$), the nesting location $\nu_p(G)$ can hide this removal to the rest of the graph (the action is not propagated to lower levels), if it adds the location L on its own. The case (iv) is analoguous, but for locations that are removed in other locations of the graph: even if the nested graph does not catch L, the nesting on can catch and create it. Cases (ii) and (iii) are a bit different; take for instance case (ii): if the nested graph G captures an L (removed by an other location), then the nesting graph *has to* forward the interaction.

Example 7 (Semantics of higher-order nesting)**.** Let reconsider the simple graph $A \parallel B \parallel C$ of the sidebox above with the same partition function p, with the addition that $p(A' \parallel B' \parallel C) = \{A' \parallel B', C\}$ and $p(A' \parallel B') = \{A', B'\}$.

$$\begin{aligned}\nu_p(A \parallel B \parallel C) = [A : \Sigma(A) \lhd p_A \bullet r_A] \\ \parallel [[B : \Sigma(B) \lhd p_B \bullet r_B] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C] : \Sigma(B) \parallel \Sigma(C) \lhd p_{BC} \bullet r_{BC}]\end{aligned}$$

where p_{ABC} and r_{ABC} are the set of unbound provided (resp. required) roles of $A' \parallel B' \parallel C$, with the addition of roles in $\mathbb{R}_2$ (an respectively for other indices).
Let illustrate how the nesting level of a location can change, thanks to the proposed semantics. So far, in $\nu_p(A \parallel B \parallel C)$, A is nested at depth 1 and B and C are nested at level 2. When $A \parallel B \parallel C$ reduces to $A' \parallel B' \parallel C$, A' and B' should belong to the same sub-aggregate, at depth 2, while C is alone at level 1. Figure 3.9 shows the transition considered in this example.
We show how both A and C have to change their depth. Notice that, as with regular nesting, there are multiple ways to remove and add locations to achieve the partitioning. We show one of those possibilities. For the sake of simplicity, we ignore all sanity checks (the conditions on the well form of the newly created graph, the conditions on the priorities and interactions, on the environment) as well as the authorisation function in the premisses of the rules, and the details of Γ and Γ_H.

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{\Gamma \vdash_{\mathcal{T}_1} B \xrightarrow{\Lambda_B} B'}{\Gamma_H \rhd [B : \Sigma(B) \lhd p_B \bullet r_B] \xrightarrow{\Lambda_B} [B' : \Sigma(B) \lhd p_{B'} \bullet r_{B'}] \in \mathcal{T}_H}\ (\mathcal{T}_H)
}{\Gamma_H \vdash_{\mathcal{T}_H} [B : \Sigma(B) \lhd p_B \bullet r_B] \xrightarrow{\Lambda_B} [B' : \Sigma(B) \lhd p_{B'} \bullet r_{B'}]}\ (\textsc{Trans})
}{\Gamma_H \vdash_{\mathcal{T}_H} [B : \Sigma(B) \lhd p_B \bullet r_B] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C] \xrightarrow{\Lambda_B} [B' : \Sigma(B') \lhd p_{B'} \bullet r_{B'}] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C]}\ (\textsc{Ctx})
}{\Gamma_H \rhd [[B : \Sigma(B) \lhd p_B \bullet r_B] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C] : \Sigma(B \parallel C) \lhd p_{BC} \bullet r_{BC}] \xrightarrow{\langle \Lambda_B.\mathtt{prior}, \Lambda_B.\mathtt{sync} \cup \{r:\overline{\mathtt{rmv}}\langle C\rangle, r:\mathtt{rmv}\langle A'\rangle\}\rangle} [[B' : \Sigma(B') \lhd p_{B'} \bullet r_{B'}] \parallel [A' : \Sigma(A') \lhd p_{A'} \bullet r_{A'}] : \Sigma(A' \parallel B') \lhd p_{A'B'} \bullet r_{A'B'}] \in \mathcal{T}_H}\ (\mathcal{T}_H)
}{\Gamma_H \vdash_{\mathcal{T}_H} [[B : \Sigma(B) \lhd p_B \bullet r_B] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C] : \Sigma(B \parallel C) \lhd p_{BC} \bullet r_{BC}] \xrightarrow{\langle \Lambda_B.\mathtt{prior}, \Lambda_B.\mathtt{sync} \cup \{r:\overline{\mathtt{rmv}}\langle C\rangle, r:\mathtt{rmv}\langle A'\rangle\}\rangle} [[B' : \Sigma(B') \lhd p_{B'} \bullet r_{B'}] \parallel [A' : \Sigma(A') \lhd p_{A'} \bullet r_{A'}] : \Sigma(A' \parallel B') \lhd p_{A'B'} \bullet r_{A'B'}]}\ (\textsc{Trans})
}{(H_1)}
$$

$$(\mathfrak{T}_H)\ \frac{\Gamma \vdash_{\mathfrak{T}_1} A \xrightarrow{\Lambda_A} A'}{(\textsc{Trans})\ \dfrac{\Gamma \rhd [A : \Sigma(A) \lhd p_A \bullet r_A] \xrightarrow{\langle \Lambda_A.\texttt{prior}, \Lambda_A.\texttt{sync} \cup \{r:\overline{\texttt{rmv}}\langle A'\rangle, r:\texttt{rmv}\langle C\rangle\}\rangle} [C : \Sigma(C) \lhd p_C \bullet r_C] \in \mathfrak{T}_H}{\dfrac{\Gamma_H \vdash_{\mathfrak{T}_H} [A : \Sigma(A) \lhd p_A \bullet r_A] \xrightarrow{\langle \Lambda_A.\texttt{prior}, \Lambda_A.\texttt{sync} \cup \{r:\overline{\texttt{rmv}}\langle A'\rangle, r:\texttt{rmv}\langle C\rangle\}\rangle} [C : \Sigma(C) \lhd p_C \bullet r_C]}{(H_2)}}}$$

$$(\textsc{Comp})\ \frac{(H_1) \qquad (H_2)}{\begin{array}{l}\Gamma_H \vdash_{\mathfrak{T}_H} [A : \Sigma(A) \lhd p_A \bullet r_A] \parallel [[B : \Sigma(B) \lhd p_B \bullet r_B] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C] : \Sigma(B \parallel C) \lhd p_{BC} \bullet r_{BC}] \\ \quad \xrightarrow{\Lambda_B} [[B' : \Sigma(B') \lhd p_{B'} \bullet r_{B'}] \parallel [A' : \Sigma(A') \lhd p_{A'} \bullet r_{A'}] : \Sigma(A' \parallel B') \lhd p_{A'B'} \bullet r_{A'B'}] \parallel [C : \Sigma(C) \lhd p_C \bullet r_C]\end{array}}$$

Chapter 4

Encapsulation policies in Location Graphs

In the introduction (Chapter 1), we showed that the notion of encapsulation and isolation was more subtle than what one can think first, and we showed that by highlighting multiple isolation schemes, which all made sense for a given context.

We propose the location graph framework and we state that it is suitable to implement various schemes. To support this proposition, this chapter shows how the policies presented in the introduction can be implemented in the location graph framework.

The example we follow in this thesis where not chosen at random, we intentionally selected policies that where quite similar (such as the variations around strict encapsulation), in order to show that the capabilities of the framework are fine grained: the small subtleties of each policies are reflected in the authorisation function. We also selected very distinct policies (such as the logging system, which is very ad-hoc, compared to the encapsulation schemes).

In the first section of this chapter, we show how the three variants of the encapsulation policy can be implemented using the location graphs. We begin with the basic strict encapsulation, used e.g. in actor models, for which we prove in details that it is correct and indeed correspond to a notion of encapsulation. We then show how, by modifying slightly the definitions, we can fall back on the variants of that policy.

In a second section, we implement our logging system example. To illustrate the flexibility of the framework, we intentionally use a very different approach. For the encapsulation based policies, we rely on the authorisation function, i.e. runtime verification, while the implementation of the logging-system rely on a careful definition of the unconstrained semantics, i.e. without runtime verification.

Contents

4.1 Hierarchical policies 73
4.1.1 Actor Model 73
4.1.2 Shared Encapsulation Policy 80
4.1.3 Multi-Level Encapsulation Policy 82
4.2 Logging system 82

4.1 Hierarchical policies

In this first section, we will present the three hierarchical policies we are interested in. First, we will take a deep look at the actor model, i.e. one level of nesting with a strict communication policy. From this in-depth explanation, we will extrapolate the variants (multiple levels of nesting and sharing), which can be achieved with some minimal adjustments.

4.1.1 Actor Model

Presentation. To provide strict encapsulation, we will mark each location as being an owner or being owned. We use the sort for this marking. Therefore, we will have two kinds of sorts. In order to identify the ownership relation, we will (i) identify each location with a special identity role; and (ii) require each owner to store in its sort the identity of the locations it owns. To avoid double ownership, the owner also binds the identity role of locations it owns. Thanks to location graph semantics, there can not be to different owners of the same location.

In this section, we use the index $[\cdot]_{se}$ (for *strict encapsulation*) to annotate elements.

Model. Given a set of processes $\mathbb{P}_{se}$, a set of roles $\mathbb{R}_{se}$, a set of value $\mathbb{V}_{se}$ and a set of channel $\mathbb{C}_{se}$, we define our set of sorts $\mathbb{S}_{se}$. We have two variants of sorts: for owned locations, we have $\langle r \rangle$, which is a

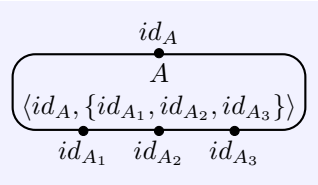


Figure 4.1: Representation of a single owner location. This location (formally written $L = [A : \langle id_A, \{id_{A_1}, id_{A_2}, id_{A_3}\}\rangle \lhd \{id_A\} \bullet \{id_{A_1}, id_{A_2}, id_{A_3}\}]$) is identified by its role id_A ($L.\texttt{id} = id_A$) and owns three locations, identified by id_{A_1}, id_{A_2} and id_{A_3} ($L.\texttt{owned} = \{id_{A_1}, id_{A_2}, id_{A_3}\}$). Notice that, according to Definition 73 above, id_A is bound in the provided direction and id_{A_1}, id_{A_2}, and id_{A_3} are bound in the required direction.

1-tuple which contains only the identifier of the location; and for owner locations, we have $\langle r, \sigma\rangle$ which contains, in addition, the set of owned identifiers.

Definition 70 (Strict encapsulation sort)**.**

$$\mathbb{S}_{se} = \{\langle r\rangle | r \in \mathbb{R}\} \cup \{\langle r, \sigma\rangle | r \in \mathbb{R}, \sigma \subset \mathbb{R}\}$$

We have the two functions $[\cdot].\texttt{id}$ and $[\cdot].\texttt{owned}$ which allow to access, for each location, its identifier and the set of owned locations ($\emptyset$ for owned locations).

Definition 71 (Identifier of a strict encapsulation sort)**.**

$$\forall s \in \mathbb{S}_{se} \cdot s.\texttt{id} \triangleq \begin{cases} r & \text{if } s = \langle r\rangle \\ r & \text{if } s = \langle r, \sigma\rangle \end{cases}$$

Definition 72 (Owned locations of a strict encapsulation sort)**.**

$$\forall s \in \mathbb{S}_{se} \cdot s.\texttt{owned} \triangleq \begin{cases} \emptyset & \text{if } s = \langle r\rangle \\ \sigma & \text{if } s = \langle r, \sigma\rangle \end{cases}$$

We want to enforce locations to bind their identifier (in provided direction) and, for owner locations, all owned identifier (in required direction). Therefore, we say that locations are well-formed (w.r.t. strict encapsulation) if the following predicate holds:

Definition 73 (Well-formed locations (w.r.t. strict encapsulation))**.**

$$\forall P \in \mathbb{P}_{se}, s \in \mathbb{S}_{se}, p \subset \mathbb{R}_{se}, r \subset \mathbb{R}_{se} \cdot \texttt{WF}_{se}([P : s \lhd p \bullet r]) \triangleq s.\texttt{id} \in p \wedge s.\texttt{owned} \subseteq r$$

We call $\mathbb{L}_{se}$ the set of all locations L such that $\texttt{WF}_{se}(L)$ holds. We call $\mathbb{G}_{se}$ the set of all location graphs G such that $\forall L \in G \cdot \texttt{WF}_{se}(L)$.

In this section, we will only consider well-formed locations, according to this predicate. This will be easily ensured by the authorisation function. Figure 4.1 shows a representation of a well-formed location of a single owner node (say L).

Finally, we define the predicate $[\cdot].\texttt{is\_owner}$ which holds if and only if the given (skeleton) location is an owner (skeleton) location.

Definition 74 (Owner predicate)**.**

$$\forall L \in \mathbb{L} \cdot L.\texttt{is\_owner} \triangleq \exists r \in \mathbb{R}_{se}, \sigma \subset \mathbb{R}_{se} \cdot L.\texttt{sort} = \langle r, \sigma\rangle$$

$$\forall L_s \in \mathbb{L}^{\mathbf{s}} \cdot L_s.\texttt{is\_owner} \triangleq \exists r \in \mathbb{R}_{se}, \sigma \subset \mathbb{R}_{se} \cdot L_s.\texttt{sort} = \langle r, \sigma\rangle$$

Remark. The function $[\cdot].\texttt{owned}$ is a function from sorts ($\mathbb{S}_{se}$) to sets of roles $\mathcal{P}(\mathbb{R}_{se})$. We extends the notation to locations and location graphs:

$$\forall L \in \mathbb{L}_{se} \cdot L.\texttt{owned} \triangleq L.\texttt{sort.owned} \qquad \forall G \in \mathbb{G}_{se} \cdot G.\texttt{owned} \triangleq \bigcup_{L \in G} L.\texttt{owned}$$

Similarly, we extend $[\cdot].\texttt{id}$ to locations.

$$\forall L \in \mathbb{L}_{se} \cdot L.\texttt{id} \triangleq L.\texttt{sort.id}$$

◁

Remark. By construction, each location L is either an *owner* (if the predicate $L.\texttt{is\_owner}$ holds) or *owned* (otherwise). ◁

Remark. For each location L, $L.\texttt{owned} \subseteq L.\texttt{required}$. Hence, by construction, for any two locations L_1 and L_2 in a graph, $L_1.\texttt{owned}$ and $L_2.\texttt{owned}$ are disjoints. ◁

Semantics. The semantics of the strict encapsulation policy is not defined using the unconstrained location transitions: we only need to specify the authorisation function.

To define the authorisation function, we first formalise the notion of ownership domain. Then, we show how we can infer, from a (Trans) rule, which location are reached. Finally, the authorisation function is defined such that it holds if and only if all location reached are within the ownership domain.

We say that L_1 owns L_2 (written $L_1 \multimap L_2$) when $L_2.\texttt{id}$ is in $L_1.\texttt{owned}$.

Definition 75 (Strict ownership relation)**.** Let $[\cdot] \multimap [\cdot]$ be the smallest relation over locations such that:

$$L_1 \multimap L_2 \Leftrightarrow L_1.\texttt{sort} = \langle \_, \sigma \rangle \wedge L_2.\texttt{sort} = \langle id_2 \rangle \wedge id_2 \in \sigma$$

Lemma 27 (Owner uniqueness)**.**

$$\forall G \in \mathbb{G}_{se}, L_o, L'_o, L \in G \cdot L_o \multimap L \wedge L'_o \multimap L \Rightarrow L_o = L'_o$$

Proof. Suppose there exists $L_o \in G$ and $L'_o \in G$ such that $L_o \neq L'_o$ and $L_o \multimap L$ and $L'_o \multimap L$.
Since $L_o \multimap L$, then $L_o.\texttt{sort} = \langle \_, \sigma \rangle \wedge L.\texttt{sort} = \langle id \rangle \wedge id \in \sigma$. In addition, from Definition 73, $L_o.\texttt{owned} = \sigma \subseteq L_o.\texttt{required}$. Therefore, $id \in L_o.\texttt{required}$.
Similarly for L'_o: $id \in L'_o.\texttt{required}$.
Therefore, $L_o.\texttt{required} \cap L'_o.\texttt{required} \neq \emptyset$. Thus $\texttt{WF}_G(G)$ does not hold, i.e. $G \notin \mathbb{G}_{se}$. Contradiction. □

The ownership domain of L is the set of all locations that are owned by L if L is an owner or that are owned by the same owner if L is an owned location.

Definition 76 (Strict ownership domain)**.** Given a graph G and a location L of G, the *strict ownership domain*[a] of L, noted $G\#L$, is defined as:

$$G\#L \triangleq \begin{cases} \{L_i \in G | L \multimap L_i \vee L = L_i\} & \text{if } L.\texttt{is\_owner} \\ \{L_i \in G | L_i \multimap L \vee (\exists L_o \in G \cdot L_o \multimap L \wedge L_o \multimap L_i)\} & \text{otherwise} \end{cases}$$

[a] We specify that the ownership domain is strict to avoid confusion with the notion of ownership domain of relaxed policies below.

Remark. Stricto sensus, $G\#L$ is a set of locations. When the context is clear, we extends the notation such that $G\#L$ also represent the location graph formed by the composition of all locations in the set: $G\#L = \prod_{L' \in G\#L} L'$ ◁

Remark. Similarly, we extend the concept to skeleton graphs and skeleton locations. ◁

Definition 77 (Strict ownership domain (excluding owner))**.** Given a graph G and a location L

of G, the *strict ownership domain excluding the owner* of L, noted $G \star L$, is defined as:

$$G \star L \triangleq \begin{cases} \{L_i \in G | L \multimap L_i\} & \text{if } L.\texttt{is\_owner} \\ \{L_i \in G | \exists L_o \in G \cdot L_o \multimap L \wedge L_o \multimap L_i\} & \text{otherwise} \end{cases}$$

Remark. Similarly, we extend the notation such that $G \star L$ also denotes the location graph formed by the composition of all locations in the set: $G \star L = \prod_{L' \in G \star L} L'$ ◁

Remark (Ownership domains of skeleton graphs). Similarly, we extend the concept to skeleton graphs and skeleton locations. Given a skeleton graph G_s and a skeleton location L_s, we note $G_s \#_s L_s$ for the ownership domain of L_s in G_s and $G_s \star_s L_s$ for the ownership domain of L_s in G_s, excluding the owner.

Since the definition of ownership domains does not depend on the processes of locations, everything works rights (i.e. $\forall G \in \mathbb{G}_{se}, L \in G \cdot \Sigma(G \# L) = \Sigma(G) \#_s \Sigma(L)$ — resp. for ownership domain excluding the owner). ◁

Remark. If L is an owner role, then $L \notin G \star L$. ◁

Lemma 28.

$$\forall G, L, L_o \cdot L_o \multimap L \Rightarrow G \# L = G \star L \cup \{L_o\}$$

Proof. Since $L_o \multimap L$, then $\neg L.\texttt{is\_owner}$. Therefore:

$$\begin{aligned} G \# L &\overset{\text{Def } 76}{=} \{L_i \in G | L_i \multimap L \vee (\exists L_o \in G \cdot L_o \multimap L \wedge L_o \multimap L_i)\} \\ &= \{L_i \in G | L_i \multimap L\} \cup \{L_i \in G | \exists L_o \in G \cdot L_o \multimap L \wedge L_o \multimap L_i\} \\ &\overset{\text{Def } 77}{=} \{L_i \in G | L_i \multimap L\} \cup G \star L \\ &\overset{\text{Lem } 27}{=} \{L_o\} \cup G \star L \end{aligned}$$

□

We now define the *range* of a transition label, which is the set of all (skeleton) locations that are affected by the label.

Definition 78 (Label range)**.** Given a skeleton graph $G_s \in \mathbb{G}_{se}$ and a label Λ, the *range* of Λ is a set of skeleton locations defined as:

$$\texttt{range}(\Lambda, G_s) = \{L_s | L_s \in G_s \wedge (L_s.\texttt{roles} \cap \Lambda.\texttt{roles} \neq \emptyset)\}$$

Finally, we show that our notion of *range* follows the intuition: all roles affected by a label are either bound to a location[1] in the range or are unbound.

Lemma 29.

$$\begin{aligned} \forall \Lambda, G_s, \Delta, L, C, r \cdot \Delta \cdot G_s \rhd L \xrightarrow{\Lambda} C \Rightarrow \\ r \in \Lambda.\texttt{roles} \Rightarrow \\ r \in G_s.\texttt{unbound} \vee \exists L_r \in (\texttt{range}(\Lambda, G_s) \setminus \Sigma(L)) \cdot r \in L_r.\texttt{bound} \end{aligned}$$

Proof. By definition of label range (Definition 78), we have to prove that $r \in G_s.\texttt{unbound} \vee \exists L_r \in (\{L_s | L_s \in G_s \wedge L_s.\texttt{roles} \cap \Lambda.\texttt{roles} \neq \emptyset\} \setminus \Sigma(L)) \cdot r \in L_r.\texttt{bound}$.
By hypothesis, $r \in \Lambda.\texttt{roles}$. By definition of the unconstrained location transition, in particular the constraints on the label, since $r \in \Lambda.\texttt{roles}$, $r \in L.\texttt{provided}$ or $r \in L.\texttt{required}$, and therefore, $r \in G_s.\texttt{roles}$ in both cases.
If a skeleton location $L' \neq L$ of G_s also binds r, then $L' \in \texttt{range}(\Lambda, G_s)$.
Otherwise, $G_s \equiv \Sigma(L) \parallel G'_s$ and $r \notin G'_s.\texttt{roles}$. Therefore, $r \in G_s.\texttt{unbound}$. □

[1] Strictly speaking, to a skeleton location.

Authorisation function. Our authorisation function ensures two aspects of the strict encapsulation policy: (i) preventing communications between owned locations that do not belong to the same ownership domains; and (ii) preserving the ownership structure (i.e. prevent orphans and exchanges of locations between ownership domains).

Concerning communications, a transition $\Gamma \rhd L \xrightarrow{\Lambda} C$ is allowed (in the context of a skeleton graph G_s) if and only if messages are exchanged (i) between locations belonging to the same ownership domain; or (ii) between owner locations. Hence, for owned locations, we require that:

$$L.\texttt{is\_owner} \Rightarrow \forall L_r \in \texttt{range}(\Lambda, G_s) \cdot L_r \in G_s \#_s \Sigma(L) \vee L_r.\texttt{is\_owner}$$

and, for owned location:

$$\neg L.\texttt{is\_owner} \Rightarrow \texttt{range}(\Lambda, G_s) \subseteq G_s \#_s \Sigma(L)$$

Concerning node creation, for the same transition $\Gamma \rhd L \xrightarrow{\Lambda} C$, if L is an owned location, it should be allowed to create locations only within the same ownership domain. However, it is not possible to ensure that a transition in which the starting location is owned creates only nodes in the same ownership domain[2]. Thus, we forbid such creation: the resulting graph should contain a single location which identifier is the same than the starting location.

$$\neg L.\texttt{is\_owner} \Rightarrow \text{size}(C) \leq 1 \wedge \forall L_c \in C, L_c.\texttt{id} = L.\texttt{id}$$

Owner locations can create new locations in their ownership domain. Also, they can get rid of locations they own, and create orphan locations. Such orphan locations can subsequently be caught by other owner location, hence allowing some sort of exchange. Therefore, we do not impose additional constraints on owner locations.

Finally, the authorisation function is the following:

$$\begin{aligned}\texttt{Auth}_{se}(G_s, \Gamma \rhd L \xrightarrow{\Lambda} C) = {} & L.\texttt{is\_owner} \Rightarrow \forall L_r \in \texttt{range}(\Lambda, G_s) \cdot L_r \in G_s \#_s \Sigma(L) \vee L_r.\texttt{is\_owner} \\ & \wedge \neg L.\texttt{is\_owner} \Rightarrow \texttt{range}(\Lambda, G_s) \subseteq G_s \#_s \Sigma(L) \\ & \wedge \neg L.\texttt{is\_owner} \Rightarrow \text{size}(C) \leq 1 \wedge \forall L_c \in C, L_c.\texttt{id} = L.\texttt{id}\end{aligned}$$

There is an interesting remark about this authorisation function[a]: its value depend only on a small subset of the skeleton graph. Informally, to decide whether an unconstrained location transition is allowed, it only needs to look at the ownership domain the location belongs to.
More formally, for any location L, there exists a skeleton graph G_s such that for any G'_s which contains G_s, $\texttt{Auth}(G_s, \Gamma \vdash_{\mathcal{T}_u} L \xrightarrow{\Lambda} C)$, if and only if $\texttt{Auth}(G'_s, \Gamma \vdash_{\mathcal{T}_u} L \xrightarrow{\Lambda} C)$ for any unconstrained location transition[b]. We call the smallest G_s the *decision subgraph of* L. For instance, for the presented strict encapsulation, the smallest decision graph of each L is $G \# L$.
This means that if a transition is taken by a graph G_1, then this same transition can also be taken whatever graph G_2 we compose G_1 with.
We say such authorisation functions are *composable*.
In addition, if we can partition the graph into subgraphs G_i such that the decision subgraph of each location L_i^j of each G_i is G_i, we call this authorisation function *independently composable*.
Finally, we say an authorisation function is *local* if it does not depend on the skeleton graph (and environment).
Intuitively, those notions are related (for instance, intuitively, an instance of a location graph with an independently composable authorisation function can be nested such that the authorisation function of the resulting 2nd order graph is local. Also, those notions may prove useful in terms of analysis (which can be performed statically), optimisation (via distributing independent subgraphs, evaluating authorisation and transition concurrently), etc. Finally, we could intuitively think that the class of local authorisation functions has some correspondence with the class of laws in the Law-Governed Interactions framework ([42]). If this is the case, then (i) our framework would be strictly more expressive than LGI; and (ii) all local authorisation functions (and therefore all

[2]More precisely, in order for L to create other owned locations, it would require the cooperation of its owner L_o so that both synchronise when L creates a location, and L_o modifies its sort in order to reflect the new ownership. Since it requires the cooperation of the owner, and for the sake of simplicity, we choose to forbid these location creation.

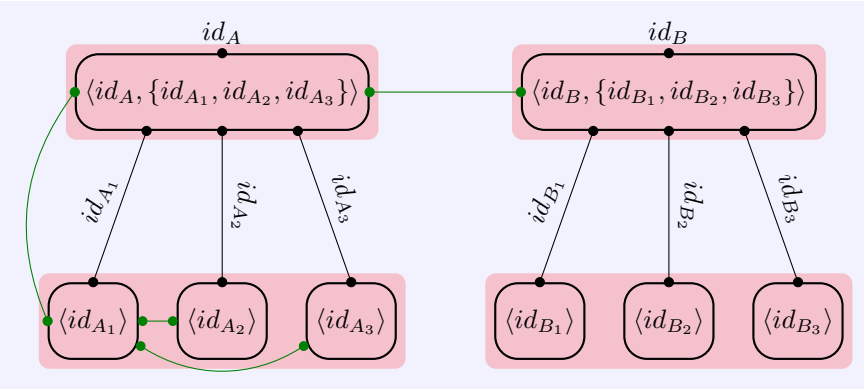


Figure 4.2: Representation of what Figure 2.1 would look like using our implementation of strict encapsulation in location graphs. Role in green represent regular roles, and those in black represent those used for identification and ownership purposes. Notice that, in our implementation, only the usage of roles (i.e. their presence in the label of a transition) is constrained, not the binding; e.g. a role from A_3 to B_1 would be allowed, but could not be used. The four areas in red shows the four elements of the partition of the graph using the partitioning function p_{se}.

composable authorisation functions, after proper nesting) could be efficiently implemented using methods developed for LGI.

[a] If we modify it slightly so that communication on unbound roles is forbidden.
[b] With an adequate Γ.

Isolation. We want to show that owned locations can not send messages to other ownership domains. We nest all owned locations belonging to the same ownership domain together, and we show that this nested ownership domain can only send messages to the owner.

We do not include the owner in the nested ownership domain, as the owner has more rights than owned locations: owner locations are allowed to exchange messages. We intent to group together locations that have the same access rights.

Definition 79 (Strict encapsulation partitioning function)**.**

$$p_{se}(G) = \{L \in G | L.\texttt{is\_owner}\} \cup \{G \star L | L \in G \wedge L.\texttt{is\_owner}\} \cup \{L | L \text{ is an orphan}\}$$

Remark. p_{se} is a graph partitioning function. Therefore, it returns a multiset of graphs[3]. ◁

To illustrate the partitioning function p_{se}, Figure 4.2 shows a location graph implementing the case presented in Figure 2.1, and how it would be partitionned using p_{se}[4].

For any graph G, $N_{p_{se}}(G)$ is a second order graph. In these second order graphs, nodes nest either: (i) a single owner; (ii) the ownership domain of a node, excluding the owner; or (iii) a single orphan owned node. Depending on the inner subgraph, we call each 2^{nd} order location (i) *owner location*; (ii) *ownership domain location*; and (iii) *orphan location*.

Informally, our goal is to prove that messages from or to an ownership domain location are from (or go to) their respective owner location. Said otherwise, we only allow communication on roles (i) between owner locations; and (ii) between an owner role and its respective ownership domain.

Theorem 5. *For any graph G, any owner location L_o in G, if $\Delta \cdot \Sigma(N_{p_{se}}(G)) \vdash_{\mathcal{T}_u} n(G \star L_o) \xrightarrow{\Lambda} G'$, then for all actions $\hat{r} : \hat{a}\langle V\rangle$ in Λ such that $\hat{r} \in \mathbb{R}$, $r \in n(L_o).\texttt{roles}$ or $r \in N_{p_{se}}(G).\texttt{unbound}$.*

Proof. First, from Lemma 4, since $\hat{r} : \hat{a}\langle V\rangle \in \Lambda$, $r \in n(G \star L_o).\texttt{unbound}$, i.e. $r \in n(G \star L_o).\texttt{roles}$. Since $\Delta \cdot \Sigma(N_{p_{se}}(G)) \vdash_{\mathcal{T}_u} n(G \star L_o) \xrightarrow{\Lambda} G'$, we have that $\texttt{Auth}(\Sigma(N_{p_{se}}(G)), \Delta \cdot \emptyset \rhd n(G \star L_o) \xrightarrow{\Lambda} G')$

[3] In this context, it actually returns a set of graph, since all locations bind at least their identifier role, and therefore, there can not be duplicates.
[4] Figure 4.3, described later, shows the corresponding second order graph.

holds. From Definition 62, we have $\Delta' \cdot e(\Sigma(N_{p_{se}}(G))) \vdash_{\mathcal{T}_1} n^{-1}(n(G \star L_o)) \xrightarrow{\texttt{prune}(\Lambda)} N_p^{-1}(G')$. From Lemma 19, $e(\Sigma(N_{p_{se}}(G))) \equiv \Sigma(G)$. Also, from Lemma 11, $n^{-1}(n(G \star L_o)) \equiv G \star L_o$. Thus:

$$\Delta' \cdot \Sigma(G) \vdash_{\mathcal{T}_1} G \star L_o \xrightarrow{\texttt{prune}(\Lambda)} N_p^{-1}(G')$$

for an adequate[a] $\Delta' \subseteq \Delta$.
Since $\hat{r} \in \mathbb{R}$, $\hat{r} : \hat{a}\langle V \rangle \in \texttt{prune}(\Lambda)$. Thus, there exists an $L \in G \star L_o$ such that $\Delta' \cdot \Sigma(G) \vdash_{\mathcal{T}_1} L \xrightarrow{\Lambda'} C'$ with $\hat{r} : \hat{a}\langle V \rangle \in \Lambda'$ (which implies, after Definition 29, that $r \in L.\texttt{roles}$). In addition, the strict encapsulation authorisation function holds: $\texttt{Auth}_{se}(\Sigma(G), \Delta' \cdot \emptyset \rhd L \xrightarrow{\Lambda'} C')$. From the remark on Definition 77, since $L \in G \star L_o$, $\neg L.\texttt{is\_owner}$; which implies $L \neq L_o$. Hence, $\texttt{range}(\Lambda', \Sigma(G)) \subseteq \Sigma(G) \#_s \Sigma(L)$. Since $L \in G \star L_o$, then $L_o \multimap L$. Therefore, from Lemma 28[b], $\Sigma(G) \#_s \Sigma(L) = (\Sigma(G) \star_s \Sigma(L)) \cup \{\Sigma(L_o)\}$, thus $\texttt{range}(\Lambda', \Sigma(G)) \subseteq (\Sigma(G) \star_s \Sigma(L)) \cup \{\Sigma(L_o)\}$.
From Lemma 29, either (a) $\exists L_r \in ((\Sigma(G) \star_s \Sigma(L)) \cup \{\Sigma(L_o)\}) \setminus \Sigma(L) \cdot r \in L_r.\texttt{roles}$; or (b) $r \in \Sigma(G).\texttt{unbound}$.
If (b) $r \in \Sigma(G).\texttt{unbound}$, then $r \in G.\texttt{unbound}$, then $r \in N_{p_{se}}(G).\texttt{unbound}$ and the conclusion follows directly.
We consider the case (a): $\exists L_r \in ((\Sigma(G) \star_s \Sigma(L)) \cup \{\Sigma(L_o)\}) \setminus \Sigma(L) \cdot r \in L_r.\texttt{roles}$.
Since $L \neq L_o$[c], $((\Sigma(G) \star_s \Sigma(L)) \cup \{\Sigma(L_o)\}) \setminus \Sigma(L) = ((\Sigma(G) \star_s \Sigma(L)) \setminus \Sigma(L)) \cup \{\Sigma(L_o)\}$. Therefore, $\exists L_r \in ((\Sigma(G) \star_s \Sigma(L)) \setminus \Sigma(L)) \cup \{\Sigma(L_o)\} \cdot r \in L_r.\texttt{roles}$.
There are two possibilities: (i) $\exists L_r \in ((\Sigma(G) \star_s \Sigma(L)) \setminus \Sigma(L)) \cdot r \in L_r.\texttt{roles}$; or (ii) $r \in \Sigma(L_o).\texttt{roles}$.
Case (ii) leads directly to the conclusion of the proof. We show that case (i) leads to a contradiction.
Case (i): $\exists L_r \in ((\Sigma(G) \star_s \Sigma(L)) \setminus \Sigma(L)) \cdot r \in L_r.\texttt{roles}$. Therefore, since $r \in L.\texttt{roles}$, $r \notin (\Sigma(G) \star_s \Sigma(L)).\texttt{unbound}$, thus $r \notin (G \star L).\texttt{unbound}$. Therefore, from Definition 49, $r \notin (n(G \star L)).\texttt{roles}$, hence, $r \notin (n(G \star L)).\texttt{unbound}$. Contradiction.

Case (ii): $r \in \Sigma(L_o).\texttt{roles}$, thus $r \in L_o.\texttt{roles}$, thus $r \in n(L_o).\texttt{roles}$.

□

[a] From the definition of the 2nd order unconstrained location transitions, $\Delta' = \Delta \cap \mathbb{R}$ where $\mathbb{R}$ is the set of roles of the 1st order graph (i.e. we remove out-of-band roles added when nesting the graph).

[b] The fact that we use skeleton ownership domains and not graph ownership domains has no influence on the result of Lemma 28.

[c] Notice that here, from $L \neq L_o$ we deduce $\Sigma(L) \neq \Sigma(L_o)$. This is true only if at least one of the sorts, the required roles or the provided roles of both locations are not equal. In our case, it is the case since each location has an identifier role bound in the provided direction (reminder: Definition 73).

Nesting. The partial bisimulation result (Theorem 4) applies in two ways: (i) first, it allows us to prove that the original (flat) graph also implement the strict encapsulation policy; and (ii) it is possible to consider each ownership domain (including the owner) as a single location, in which communications are allowed only via some roles.

Corollary 5. For any graph G, any owned location L, any location L_o such that $L_o \multimap L$, if $\Delta \cdot \Sigma(G) \vdash_{\mathcal{T}_u} L \xrightarrow{\Lambda} C$, then for all actions $\hat{r} : \hat{a}\langle V \rangle$ in Λ, either (i) $r \in L_o.\texttt{roles}$; (ii) $\exists L' \cdot L_o \multimap L' \wedge r \in L'.\texttt{roles}$; or (iii) $r \in G.\texttt{unbound}$.

Proof. Suppose none of the three possibilities hold. We show that, when nesting G with the partition function above, this leads to a contradiction. We have a location $L_2 = n(G \star L)$ in $N_{p_{se}}(G)$. Since the subgraph $G \star L$ reduces, and from Theorem 4, $n(G \star L)$ takes a similar transition $\Gamma \vdash_{\mathcal{T}_u} n(G \star L) \xrightarrow{\Lambda'} G'_2$. Since (ii) does not hold, the action $\hat{r} : \hat{a}\langle V \rangle$ in Λ is not an internal action of $n(G \star L)$. Thus, Λ' contains the action $\hat{r} : \hat{a}\langle V \rangle$.
From Lemma 5, we deduce that either $r \in n(L_o).\texttt{roles}$ (hence would (i) hold), or $r \in N_{p_{se}}(G).\texttt{unbound}$ (hence would (iii) hold). Contradiction. □

Corollary 6. Let $p_o(G) = \{G \# L_o | L_o.\texttt{is\_owner}\} \cup \{L | L \text{ is an orphan}\}$. $N_{p_o}(G)$ simulates G.

Proof. Direct from Theorem 4. □

To conclude this section, we show in Figure 4.3 and Figure 4.4 two nestings of Figure 4.2. In Figure 4.3,

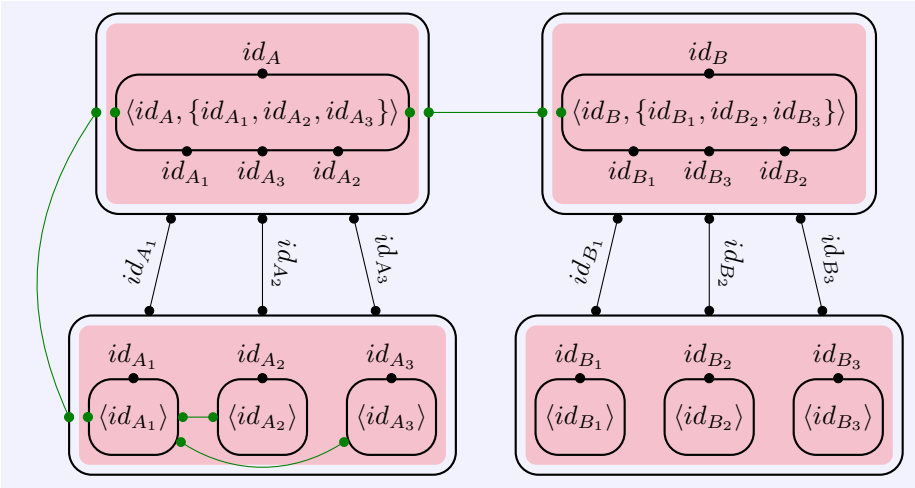


Figure 4.3: Skeleton graph of a nested location graph which implements the strict encapsulation policy using the partitioning function p_{se}. Being a skeleton graph and not the full graph, only sorts are shown in nodes and not the processes. Compare this figure with Figure 4.2: the sort of each location is the skeleton of the corresponding element of the partition (in red) formed by p_{se}. The process of each location (not shown in the skeleton graph) would be the actual subgraph.

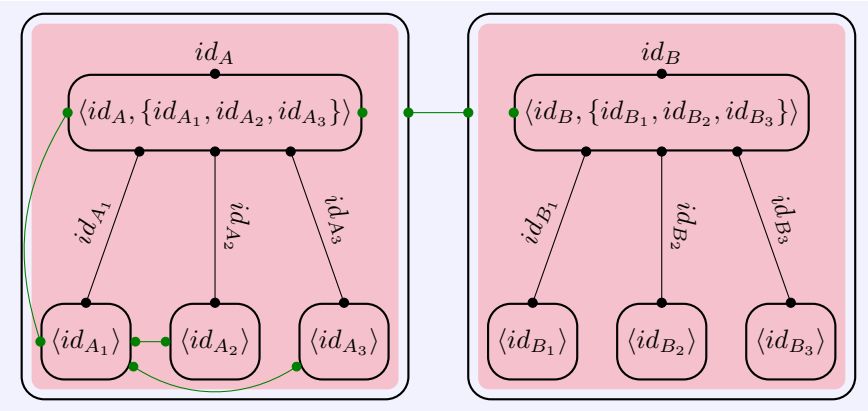


Figure 4.4: Skeleton graph of a nested location graph which implements the strict encapsulation policy using the partitioning function p_o. This partitioning function corresponds to the intuitive notion of encapsulation better than p_{se} (Figure 4.3). Nonetheless, our results show that both are equivalent, and that isolation is properly achieved.

the partitioning function used for the nesting in p_{se} which we used to show our isolation result. We see that the owner and its ownership domain are not in the same location. In Figure 4.4, we show the more intuitive nesting, in which ownership domains are nested separately together with their owner. Thank to Theorem 2, both second order graphs are similar to the original one.

4.1.2 Shared Encapsulation Policy

In Section 4.1.1, sorts are elements of the set $\mathbb{S} = \{\langle r\rangle | r \in \mathbb{R}\} \cup \{\langle r, \sigma\rangle | r \in \mathbb{R}, \sigma \subset \mathbb{R}\}$. When the sort has the form $\langle r\rangle$, the location is owned by the location that binds r.

We achieve shared encapsulation by relaxing the sort of owned location: the idea remains the same, but the sort of owned locations is now a set of roles (bound to owner locations). Hence we have $\mathbb{S} = \{\langle \sigma\rangle | \sigma \subset \mathbb{R}\} \cup \{\langle r, \sigma\rangle | r \in \mathbb{R}, \sigma \subset \mathbb{R}\}$. Figure 4.5 shows a representation of an owned location. Naturally, we also have to adapt Definition 71 to take into account that, with shared encapsulation, owned locations have multiple identifiers. For the sake of simplicity, we do not explicit the new definition.

Semantics. The semantics of the shared encapsulation policy is quite similar to those of the strict encapsulation policy. The difference lies in that a location can belong to multiple ownership domains at the same time.

The *owns* relation ($\multimap$) should be adapted to take into account the new sort of owned locations:

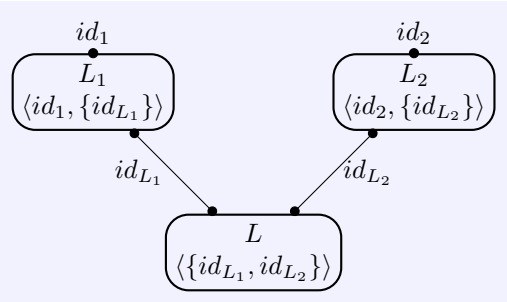


Figure 4.5: Representation of a location graph which implements the shared ownership policy. Roles id_{L_1} are id_{L_2} are bound in L and in L_1 (resp. L_2): both L_1 and L_2 own L. Notice that it is possible that a single owner location owns a owned location via multiple roles.

Definition 80 (Shared ownership relation)**.** Let $\multimap$ be the smallest relation over locations such that:

$$L_1 \multimap L_2 \Leftrightarrow L_1.\texttt{sort} = \langle \_, \sigma_1\rangle \wedge L_2.\texttt{sort} = \langle \sigma_2\rangle \wedge \sigma_1 \cap \sigma_2 \neq \emptyset$$

Notice that our definition of strict ownership domain (Definition 76) is, unexpectedly, suitable for the shared ownership case: if a location L has two owners (say L_o and L'_o), then all locations owned by L_o and L'_o are in $G\#L$.

Remark. Contrary to the strict encapsulation policy, two locations in a given ownership domain do not necessary have the same ownership domain. More formally: the following no longer holds:

$$\forall G \in \mathbb{G} \cdot \forall L \in G \cdot \forall L' \in G\#L \cdot G\#L = G\#L'$$

◁

As a consequence, the authorisation function $\texttt{Auth}_{se}$ is also suitable to ensure the shared ownership policy.

Remark. The similarities between the strict ownership policy and the shared ownership policy emphasize that strict ownership is just a special case of shared ownership, in which locations have a single owner; yet, there is no fundamental differences between the two policies. ◁

Example 8. Consider the following graph, which contains three locations with processes A, B, and C. The first two share the ownership of the last one.

$$[A : \langle id_A, \{id^1_C\}\rangle \lhd \{id_A\} \bullet \{id^1_C\}] \parallel [B : \langle id_B, \{id^2_C\}\rangle \lhd \{id_B\} \bullet \{id^2_C\}]$$

$$\parallel [C : \langle \{id^1_C, id^2_C\}\rangle \lhd \{id^1_C, id^2_C\} \bullet \emptyset]$$

Location B can create a new location D, and take ownership of that new location. Notice, in this example, that this new location is created directly with two identifiers.

$$\Gamma \rhd [B : \langle id_B, \{id^2_C\}\rangle \lhd \{id_B\} \bullet \{id^2_C\}] \xrightarrow{\langle\emptyset,\emptyset\rangle}$$

$$[B : \langle id_B, \{id^2_C, id^2_D\}\rangle \lhd \{id_B\} \bullet \{id^2_C, id^2_D\}] \parallel [D : \langle \{id^1_D, id^2_D\}\rangle \lhd \{id^1_D, id^2_D\} \bullet \emptyset]$$

Simultaneously, location A can take a shared ownership of D, using id^1_D:

$$\Gamma \rhd [A : \langle id_A, \{id^1_C\}\rangle \lhd \{id_A\} \bullet \{id^1_C\}] \xrightarrow{\langle\emptyset,\emptyset\rangle} [A : \langle id_A, \{id^1_C, id^1_D\}\rangle \lhd \{id_A\} \bullet \{id^1_C, id^1_D\}]$$

These two transitions result in a new graph in which D is added and owned by A and B:

$$[A : \langle id_A, \{id^1_C, id^1_D\}\rangle \lhd \{id_A\} \bullet \{id^1_C, id^1_D\}] \parallel [B : \langle id_B, \{id^2_C, id^2_D\}\rangle \lhd \{id_B\} \bullet \{id^2_C, id^2_D\}]$$

$$\parallel [C : \langle \{id^1_C, id^2_C\}\rangle \lhd \{id^1_C, id^2_C\} \bullet \emptyset] \parallel [D : \langle \{id^1_D, id^2_D\}\rangle \lhd \{id^1_D, id^2_D\} \bullet \emptyset]$$

4.1.3 Multi-Level Encapsulation Policy

An other extension of the strict encapsulation policy is to allow multiple levels of encapsulation. The implementation presented in Section 4.1.1 provides a suitable starting point for this policy. Two points are to be considered: (i) there shall not be differences between owner and owned locations[5], as any location can be both *owner* and *owned* at the same time; and (ii) cycles in the hierarchy shall be prevented.

To tackle these two issues, we propose the following: the set of sort is $\mathbb{S} = \{\langle r, \sigma\rangle | r \in \mathbb{R}, \sigma \subset \mathbb{R}\}$ and the set $\mathbb{R}$ of role is equipped with a partial order relation $\leq_{\mathbb{R}}$[6]. In addition, the following invariant on sorts is maintained:

$$\mathtt{I}_{mle}(\langle r, \sigma\rangle) \Leftrightarrow \forall r' \in \sigma \cdot r \leq_{\mathbb{R}} r'$$

The ownership relation ($\multimap$) is similar to the one of strict encapsulation[7]:

Definition 81 (Multi-level ownership relation)**.** Let $\multimap$ be the smallest relation over locations such that:

$$L_1 \multimap L_2 \Leftrightarrow L_1.\mathtt{sort} = \langle \_, \sigma\rangle \wedge L_2.\mathtt{sort} = \langle id_2, \_\rangle \wedge id_2 \in \sigma$$

The notions of ownership domain remains suitable for the multi-level encapsulation policy. As a consequence, the authorisation function $\mathtt{Auth}_{se}$ is also suitable to ensure the multi-level encapsulation policy.

We see that for the three encapsulation variants we presented, the authorisation function is the same, up to the definition of $\multimap$.
Future work could study further this class of authorisation function, abstracting away the relation $\multimap$ (as a parameter of the authorisation function).

4.2 Logging system

This section aims to show that the location graph framework is suitable to implement ad-hoc systems. We show how a system such as the logging system presented in the introduction (Section 1.2).

In order to illustrate the capabilities of the framework, we take a different approach than when implementing ownership systems. For ownership systems, the authorisation function dynamically checked that everything was correct, while the set of unconstrained location transition was not specified[8]. Here, on the other hand, we let the authorisation function as unspecified as possible[9], while we work on the set of unconstrained location transitions.

The choice to use the set of unconstrained location transitions instead of the authorisation function to enforce our policy is *not* a change in the framework. Instead, it shows the our framework can implement (at least some) policies in various ways.
More generally, if a policy can be enforced using only the set of unconstrained location transitions (say $\mathcal{T}_p$), then it is possible to perform the verification that a given transition is in the authorisation function (i.e. we define $\mathtt{Auth}(G_s, t) \triangleq t \in \mathcal{T}_p$) and to relax the set of uncontrained location transitions used (i.e. use a greater $\mathcal{T}'_p$ instead of $\mathcal{T}_p$).
We would trivially have the following property:

Conjecture 2. *For all graphs G_1 and G_2, for all labels Λ, the two following statements are equivalent:*

1. $\Gamma \vdash_{\mathcal{T}_p} G_1 \xrightarrow{\Lambda} G_2$ *using a trivial authorisation function; and*
2. $\Gamma \vdash_{\mathcal{T}'_p} G_1 \xrightarrow{\Lambda} G_2$ *using* $\mathtt{Auth}(G_s, t) \triangleq t \in \mathcal{T}_p$ *as the authorisation function.*

[5] As explained below, we can reuse the *owner* sorts $\langle r, \sigma\rangle$ of the strict encapsulation policy, which are composed of an identifier r and a set of owned locations σ.
[6] One can annotate roles with integer.
[7] We only need to adapt the sort of the owned location, which now has the form $\langle r, \sigma\rangle$.
[8] More precisely, the set of unconstrained location transitions in ownership based approaches is the set of all possible transitions.
[9] That is, we do as few runtime check as possible.

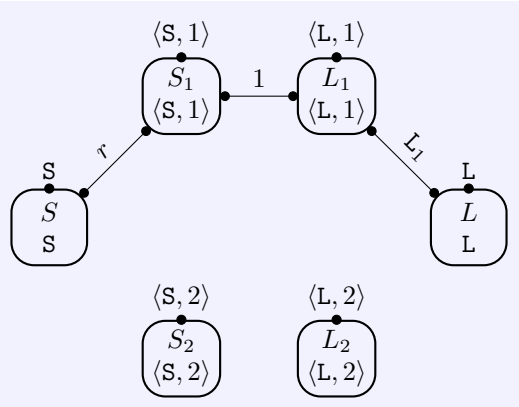


Figure 4.6: Representation of the system presented in Figure 1.6 implemented using our location graph model. Notice that there is no requirement for a locations $\langle \mathtt{S}, i\rangle$ and $\langle \mathtt{L}, i\rangle$ to bind i (e.g. between $\langle \mathtt{S}, 2\rangle$ and $\langle \mathtt{L}, 2\rangle$ here), nor for $\langle \mathtt{L}, i\rangle$ and $\mathtt{L}$ to bind L_i. Our invariant is that only these locations can bind it, and that it is the only role they can bind together. The role r is taken arbitrarily from $\mathbb{R}_b$.

As a matter of fact, one should try to define the set of unconstrained location transition as small as possible. By doing so, it might be possible to make additional assumptions and therefore reduce the amount of verifications the authorisation function has to do (which, in a software engineering perspective) are most expensive to perform, since it requires to freeze the global skeleton graph.

Graph model. Locations either belong to the main system or to the logging part of the whole system. Also, for each subcomponent of the main system, there is a corresponding logger. We use two symbols ($\mathtt{L}$ and $\mathtt{S}$) in the sort to specify in which part each location belongs. Also, each location has an integer, which is used to associate each subcomponent of the main subsystem with its logging component. Finally, we distinguish two sorts without integer (i.e. just $\mathtt{S}$ and $\mathtt{L}$) for the locations that implement each subsystem entrypoint.

Definition 82 (Logging system sorts)**.**

$$\mathbb{S}_{ls} \triangleq \{\langle t, n\rangle | t \in \{\mathtt{S}, \mathtt{L}\} \wedge n \in \mathbb{N}\} \cup \{\mathtt{S}, \mathtt{L}\}$$

Components of the main subsystem can freely communicate together (and with the main component entry point). In addition, each component is allowed to send messages to its corresponding logger, which can transfer it to the logger entrypoint. We ensure that by constraining which roles each location can bind. We consider a set $\mathbb{R}_b$ of basic role names. Those are used among components of the main subsystem.

For communication between main components and their respective logging subcomponent, we use role names taken among $\mathbb{N}$, where each component can only bind the role which correspond to the index in its sort[10].

For the communication between logger components and the logger entrypoint, logging subsystem $\langle \mathtt{L}, i\rangle$ can bind a role $\mathtt{L}_i$ toward the logger entrypoint. We call $\mathbb{L}_i$ the set of all $\mathtt{L}_i$ for $i \in \mathbb{N}$.

Notice that, a logger component with sort $\langle \mathtt{L}, i\rangle$ can only bind the roles $\mathtt{L}_i$ and i (and their identifier role $\langle \mathtt{L}, i\rangle$). This prevent two logger components to interact, as long as they do not have the same identifier.

To prevent duplicate identifiers, and similarly to identifier roles in ownership systems, we require each location to bind its sort as a special identifier role.

Definition 83 (Logging system roles)**.**

$$\mathbb{R}_{ls} \triangleq \mathbb{R}_b \cup \mathbb{N} \cup \{\mathtt{L}_i | i \in \mathbb{N}\} \cup \mathbb{S}_{ls}$$

Let $\mathbb{G}_{ls}$ be the set of location graphs formed with the above mentioned sorts and roles.

[10]Note, as a side effect, that neither the logging and the main system entrypoints can bind such roles.

Semantics. Concerning messages, we suppose we are given a set $\mathbb{M}$ of messages that components intend to exchange. To those messages, we add a set of administrative messages for the synchronisation of component deletion[11]; which we suppose distinct from messages in $\mathbb{M}$. This family of messages is the set $\{\texttt{remove}(i) | i \in \mathbb{N}\}$.

When a main subsystem component $\langle \texttt{S}, i\rangle$ is removed, it emits $\texttt{remove}(i)$ toward its respective logger component $\langle \texttt{L}, i\rangle$, which is matched only if the logger is also removed during the same transition.

We can now define our set of unconstrained location transitions.

Definition 84 (Unconstrained transitions of the logging system)**.**

$$
\begin{aligned}
\mathcal{T}_{ls} \triangleq \{\Gamma \rhd L \xrightarrow{\Lambda} C | \forall L' \in C \cdot & L'.\texttt{sort} \in L'.\texttt{provided} \\
& \wedge L'.\texttt{sort} = \texttt{L} \Rightarrow L'.\texttt{roles} \subset \{\langle \texttt{L}, i\rangle\} \cup \{\texttt{L}\} \\
& \wedge L'.\texttt{sort} = \texttt{S} \Rightarrow L'.\texttt{roles} \subset \mathbb{R}_b \cup \{\texttt{S}\} \\
& \wedge L'.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow L'.\texttt{roles} \subset \mathbb{R}_b \cup \{i\} \cup \{\langle \texttt{S}, i\rangle\} \wedge i \in L.\texttt{provided} \\
& \wedge L'.\texttt{sort} = \langle \texttt{L}, i\rangle \Rightarrow L'.\texttt{roles} \subseteq \{i\} \cup \{\langle \texttt{L}, i\rangle\} \cup \{\texttt{L}_i\} \wedge i \in L.\texttt{required} \\
\wedge\ & L.\texttt{sort} = \texttt{L} \Rightarrow \neg C \equiv \emptyset \\
\wedge\ & L.\texttt{sort} = \texttt{S} \Rightarrow \neg C \equiv \emptyset \\
\wedge\ & L.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow (C \equiv \emptyset \Leftrightarrow \Lambda.\texttt{sync} = \{i : \texttt{rmv}\langle \texttt{remove}(i)\rangle\}) \\
\wedge\ & L.\texttt{sort} = \langle \texttt{L}, i\rangle \Rightarrow (C \equiv \emptyset \Leftrightarrow \Lambda.\texttt{sync} = \{\overline{i} : \overline{\texttt{rmv}}\langle \texttt{remove}(i)\rangle\}) \\
\wedge\ & \neg C \equiv \emptyset \Rightarrow (\exists L' \in C \cdot L.\texttt{sort} = L'.\texttt{sort}) \\
\wedge\ & \forall L' \in C \cdot (L'.\texttt{sort} \neq L.\texttt{sort}) \Rightarrow \\
& \qquad (L'.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow \exists L'' \in C \cdot L''.\texttt{sort} = \langle \texttt{L}, i\rangle) \\
& \qquad \wedge (L'.\texttt{sort} = \langle \texttt{L}, i\rangle \Rightarrow \exists L'' \in C \cdot L''.\texttt{sort} = \langle \texttt{S}, i\rangle) \\
\}&
\end{aligned}
$$

Correctness. We have a notion of well-formness for our system: intuitively, a location graph implementing this logging system pattern is well-formed if and only if: (i) there is a one-to-one mapping between components of the main system and logger components; and (ii) there is no exchange between unmatched component/logger, nor among logger components.

Definition 85 (Well-formed logging system)**.**

$$
\begin{aligned}
\texttt{WF}(G) \triangleq & \forall i \in \mathbb{N} \cdot (\exists L_s \in G \cdot L_s.\texttt{sort} = \langle \texttt{S}, i\rangle) \Leftrightarrow (\exists L_s \in G \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle) \\
& \wedge \forall L \in G \cdot L.\texttt{sort} \in L.\texttt{provided} \\
& \wedge \forall L \in G \cdot (L.\texttt{sort} = \texttt{L} \Rightarrow L.\texttt{roles} \subset \{\langle \texttt{L}, i\rangle\} \\
& \qquad \wedge L.\texttt{sort} = \texttt{S} \Rightarrow L.\texttt{roles} \subset \mathbb{R}_{ls} \\
& \qquad \wedge L.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow L.\texttt{roles} \subset \mathbb{R}_{ls} \cup \{i\} \wedge i \in L.\texttt{provided} \\
& \qquad \wedge L.\texttt{sort} = \langle \texttt{L}, i\rangle \Rightarrow L.\texttt{roles} \subseteq \{i\} \cup \{\langle \texttt{L}, i\rangle\} \wedge i \in L.\texttt{required})
\end{aligned}
$$

This definition has three independent clauses: (i) the first line states that subcomponents and logger components are one-to-one mapped; (ii) the second states that each sort is used at most once; and (iii) finally, the third does the hard work: it regulates communications via role constraints.

To show our implementation is correct, we simply show that, using our unconstrained location transitions, the set of well-formed graph is closed under location graph reduction. We prove this by proving the three conditions of well-formness, in three separate lemmas.

Lemma 30.

$$\forall G \in \mathbb{G}_{ls} \cdot \forall i \in \mathbb{N} \cdot \texttt{WF}(G) \Rightarrow i \notin G.\texttt{unbound}$$

Proof. Since $\texttt{WF}(G)$, then $(\exists L_s \in G \cdot L_s.\texttt{sort} = \langle \texttt{S}, i\rangle) \Leftrightarrow (\exists L_l \in G \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle)$. Also, again

[11] Remember that, when a subcomponent of the main subsystem is remove, we want to simultaneously remove the associated logger component.

from the definition of $\texttt{WF}(G)$, for all $L \in G$: $L.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow i \in L.\texttt{provided}$ and $L.\texttt{sort} = \langle \texttt{L}, i\rangle \Rightarrow i \in L.\texttt{required}$).
Also, only locations with sort $\langle \texttt{S}, i\rangle$ and $\langle \texttt{L}, i\rangle$ can bind i.
Therefore $(\exists L_s \in G \cdot i \in L.\texttt{provided}) \Leftrightarrow (\exists L_l \in G \cdot i \in L.\texttt{required})$, i.e. $i \notin G.\texttt{unbound}$. □

Lemma 31.

$$\forall G \in \mathbb{G}_{ls} \cdot \texttt{WF}(G) \wedge \Gamma \vdash_{\mathcal{T}_{ls}} G \xrightarrow{\Lambda} G' \Rightarrow \forall i \in \mathbb{N} \cdot (\exists L_s \in G' \cdot L_s.\texttt{sort} = \langle \texttt{S}, i\rangle) \Leftrightarrow (\exists L_s \in G' \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle)$$

Proof. By contradiction: suppose there is an $i \in \mathbb{N}$ such that $\exists L_s \in G' \cdot L_s.\texttt{sort} = \langle \texttt{S}, i\rangle$ and such that $\neg\exists L_l \in G' \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle$[a].
Two cases are possibles: (i) there are locations L_s and L_l with sorts $\langle \texttt{S}, i\rangle$ and $\langle \texttt{L}, i\rangle$ in G ; or (ii) there are no two locations with sorts $\langle \texttt{S}, i\rangle$ and $\langle \texttt{L}, i\rangle$ in G . Notice that, since $\texttt{WF}(G)$, it is not possible to have one of the two.
We analyse the two cases separately:

Case (i): Since $\neg\exists L_l \in G' \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle$, we have that $G \equiv L_l \parallel G_l$ and, necessarily, $\Gamma \vdash_{\mathcal{T}_{ls}} L_l \xrightarrow{\Lambda_l} C_l$ such that $\neg\exists L'_l \in C_l \cdot L'_l.\texttt{sort} = \langle \texttt{L}, i\rangle$.

From the premisses of (Trans), $\Gamma.\texttt{names} \cdot \emptyset \rhd L_l \xrightarrow{\Lambda_l} C_l \in \mathcal{T}_{ls}$. Suppose $C_l \neq \emptyset$, by definition of $\mathcal{T}_{ls}$, $\exists L' \in C_l \cdot L_l.\texttt{sort} = L'.\texttt{sort}$, which contradicts the statement above. Therefore $C_l = \emptyset$.
Since $L_l.\texttt{sort} = \langle \texttt{L}, i\rangle$ and $C_l = \emptyset$, by definition of $\mathcal{T}_{ls}$, $\Lambda_l.\texttt{sync} = \{\bar{i} : \overline{\texttt{rmv}}\langle \texttt{remove}(i)\rangle\}$.
Since $i \notin G.\texttt{unbound}$, then $\bar{i} : \overline{\texttt{rmv}}\langle \texttt{remove}(i)\rangle \notin \Lambda.\texttt{sync}$, according to Lemma 4. Therefore, by definition of $\texttt{seval}(\cdot)$, there is a location $L \in G_l$ that takes a transition $\Gamma' \rhd L \xrightarrow{\Lambda} L'$ with an action $i : \texttt{rmv}\langle \texttt{remove}(i)\rangle \in \Lambda$. Since L_s binds i, $L = L_s$.

According to the definition of $\mathcal{T}_{ls}$, and since $L_s.\texttt{sort} = \langle S_i\rangle$, for all $\Gamma' \rhd L_s \xrightarrow{\Lambda_s} C_s$, $i : \texttt{rmv}\langle \texttt{remove}(i)\rangle \in \Lambda_s.\texttt{sync} \Rightarrow C_s \equiv \emptyset$.
Since $\texttt{WF}(G)$, then sorts are uniquely used (from the requirement that $\forall L \in G \cdot L.\texttt{sort} \in L.\texttt{provided}$). Also, since L_s is also removed from G, then there must be a newly created location in G' with sort $\langle \texttt{S}, i\rangle$, while no location with sort $\langle \texttt{L}, i\rangle$ is created. We fall back on the case (ii).

Case (ii): We show that it is not possible that a location with sort $\langle \texttt{S}, i\rangle$ is created while no location with sort $\langle \texttt{L}, i\rangle$ is.
Since a location L_s with sort $\langle \texttt{S}, i\rangle$ is created, there is a location L_c which takes a transition $\Gamma_c \rhd L_c \xrightarrow{\Lambda_c} C_c \in \mathcal{T}_{ls}$ with $L_s \in C_c$.
According to the definition of $\mathcal{T}_{ls}$, $\forall L' \in C_c \cdot (L'.\texttt{sort} \neq L_c.\texttt{sort}) \Rightarrow (L'.\texttt{sort} = \langle \texttt{S}, i\rangle \Rightarrow \exists L'' \in C_c \cdot L''.\texttt{sort} = \langle \texttt{L}, i\rangle)$. Therefore, $\exists L_l \in C_c \cdot L_l.\texttt{sort} = \langle \texttt{L}, i\rangle$.

□

[a] By symmetry, the same reasoning applies if $\langle \texttt{L}, i\rangle$ exists and $\langle \texttt{S}, i\rangle$ does not.

Lemma 32.

$$\forall G \in \mathbb{G}_{ls} \cdot \texttt{WF}(G) \wedge \Gamma \vdash_{\mathcal{T}_{ls}} G \xrightarrow{\Lambda} G' \Rightarrow \forall L \in G' \cdot L.\texttt{sort} \in L.\texttt{provided}$$

Proof. By definition of $\mathcal{T}_{ls}$, $\forall \Gamma \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_{ls} \cdot \forall L' \in C \cdot L'.\texttt{sort} \in L'.\texttt{provided}$. In particular if $L.\texttt{sort} \in L.\texttt{provided}$.
After Lemma 10 with $\mathcal{P}(L) \triangleq L.\texttt{sort} \in L.\texttt{provided}$, we have that:

$$\forall G, G' \in \mathbb{G}_{ls} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G' \Rightarrow (\forall L \in G \cdot L.\texttt{sort} \in L.\texttt{provided}) \Rightarrow \forall L \in G' \cdot L.\texttt{sort} \in L.\texttt{provided}$$

By hypothesis, $\forall G, G' \in \mathbb{G}_{ls} \cdot \Gamma \vdash_{\mathcal{T}_u} G \xrightarrow{\Lambda} G'$ holds. Also, since $\texttt{WF}(G)$, $\forall L \in G \cdot L.\texttt{sort} \in L.\texttt{provided}$.
Therefore, $\forall L \in G' \cdot L.\texttt{sort} \in L.\texttt{provided}$. □

Lemma 33. *For all graph $G \in \mathbb{G}_{ls}$, if $\mathtt{WF}(G)$ and $\Gamma \vdash_{\mathcal{T}_{ls}} G \xrightarrow{\Lambda} G'$, then*

$$\begin{aligned}\forall L \in G' \cdot (&L.\mathtt{sort} = \mathtt{L} \Rightarrow L.\mathtt{roles} \subset \{\langle \mathtt{L}, i\rangle\}\\ &\wedge L.\mathtt{sort} = \mathtt{S} \Rightarrow L.\mathtt{roles} \subset \mathbb{R}_{ls}\\ &\wedge L.\mathtt{sort} = \langle \mathtt{S}, i\rangle \Rightarrow L.\mathtt{roles} \subset \mathbb{R}_{ls} \cup \{i\} \wedge i \in L.\mathtt{provided}\\ &\wedge L.\mathtt{sort} = \langle \mathtt{L}, i\rangle \Rightarrow L.\mathtt{roles} \subseteq \{i\} \cup \{\langle \mathtt{L}, i\rangle \wedge i \in L.\mathtt{required}\})\end{aligned}$$

Proof. To improve readability, we write

$$\begin{aligned}\mathcal{Q}(L) \triangleq &L.\mathtt{sort} = \mathtt{L} \Rightarrow L.\mathtt{roles} \subset \{\langle \mathtt{L}, i\rangle\}\\ &\wedge L.\mathtt{sort} = \mathtt{S} \Rightarrow L.\mathtt{roles} \subset \mathbb{R}_{ls}\\ &\wedge L.\mathtt{sort} = \langle \mathtt{S}, i\rangle \Rightarrow L.\mathtt{roles} \subset \mathbb{R}_{ls} \cup \{i\} \wedge i \in L.\mathtt{provided}\\ &\wedge L.\mathtt{sort} = \langle \mathtt{L}, i\rangle \Rightarrow L.\mathtt{roles} \subseteq \{i\} \cup \{\langle \mathtt{L}, i\rangle\} \wedge i \in L.\mathtt{required})\end{aligned}$$

By definition of $\mathcal{T}_{ls}$, $\forall \Gamma \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_{ls} \cdot \forall L' \in C \cdot \mathcal{Q}(L')$. Therefore, we can introduce an application: $\forall \Gamma \rhd L \xrightarrow{\Lambda} C \in \mathcal{T}_{ls} \cdot \forall L' \in C \cdot \mathcal{Q}(L) \Rightarrow \mathcal{Q}(L')$. Therefore, after Lemma 10, $\forall G, G' \in \mathbb{G}_{ls} \cdot \Gamma \vdash_{\mathcal{T}_{ls}} G \xrightarrow{\Lambda} G' \Rightarrow (\forall L \in G \cdot \mathcal{Q}(L)) \Rightarrow (\forall L \in G' \cdot \mathcal{Q}(L))$.
By hypothesis, both $G \vdash_{\mathcal{T}_{ls}} \Lambda \xrightarrow{G'}$ and $\forall L \in G \cdot \mathcal{Q}(L)$ hold, therefore $\forall L \in G' \cdot \mathcal{Q}(L)$. □

Theorem 6.

$$\forall G \in \mathbb{G}_{ls} \cdot \mathtt{WF}(G) \wedge \Gamma \vdash_{\mathcal{T}_{ls}} G \xrightarrow{\Lambda} G' \Rightarrow \mathtt{WF}(G')$$

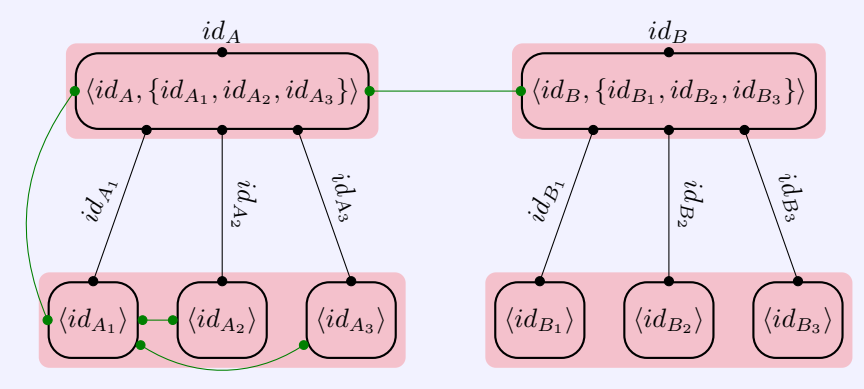
Proof. We prove separately the three conditions of the well-form predicate in Lemmas 31, 32 and 33.

□

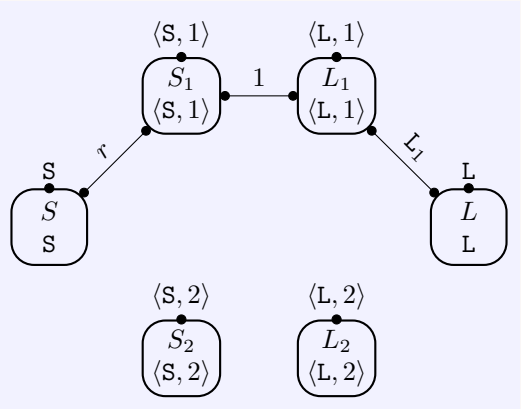
Remark. Notice that the definition of well-formed graphs does not explicitely state that there are not unwanted communications. However, it is a direct consequence of the roles each location can bind. ◁

Summary – Encapsulation policies

In this chapter, we showed, in practice, how the location graph framework can be used to implement various encapsulation policies, as well as a proof method.
For ownership-based approaches, we chosed to rely on the authorisation function, which allow very few modifications on the graph (with respect to the intuitive object graph).



For our ad-hoc example, we chose to rely on the set of unconstrained location transitions. We put more structure on the sorts and on the role identifiers.


⟨S, 1⟩
⟨L, 1⟩
S_1
⟨S, 1⟩
1
L_1
⟨L, 1⟩
r
L_1
S
L
S
S
L
L
⟨S, 2⟩
⟨L, 2⟩
S_2
⟨S, 2⟩
L_2
⟨L, 2⟩

Chapter 5

Rust implementation of the location graph framework

The location graph framework presents a convenient theoretical basis to think about complex and distributed systems. In this chapter, we show that location graph are not only a theoretical framework, but also a practical way to build actual software. To achieve this objective, we implemented a library for the Rust programming language [47] which provides a location graph framework to Rust.

Rust was chosen for convenience: its type system, based on the very ownership concepts presented in the introduction, makes both concurrent programming *easy*; and prevent accidental sharing of references, which is useful to write such a library which claims to ensure isolation.

In the first section of this chapter, we informally present the programming model (i.e. the assumptions about Rust and the main lines of the design of our implementation). Then, we present an abstract machine for this library, which provides a formal basis to reason about the library.

With the abstract machine in mind, we continue by presenting the library from the user perspective: Section 5.3 presents the programmers API, using a small example. Once familiar with the different elements of the library, the fourth section shows the internal details of the implementation of this abstract machine.

We conclude with two ancillary sections: the first shows the implementation of various utilities that can be implemented using the library, but which proved generic enough to be included within the library itself; the second shows the implementation of two of the examples taken from our bestiary.

Contents

5.1 Programming model 90
5.1.1 Design choices 90
5.1.2 Divergences from the theoretical framework 92
5.2 An abstract machine for location graphs 94
5.2.1 Preliminary definitions 94
5.2.2 Transitions, locations and local semantics 94
5.2.3 Graphs and global semantics 101
5.3 Rust API 109
5.3.1 Preliminary steps 110
5.3.2 Locations 110
5.3.3 Authorisation functions and unconstrained location transitions 111
5.3.4 Final steps 113
5.4 Implementation 115
5.4.1 Locations and Transitions 115
5.4.2 Skeleton graphs 118
5.4.3 Roles management and message exchanges 119
5.4.4 Transition selection and resolution 121
5.4.5 Authorisation functions and unconstrained location transitions set 127
5.5 Utilities 127
5.5.1 Trivial authorisation function and transition set 130
5.5.2 Generic role names 133
5.5.3 TCP connections 133

5.6 Encapsulation policies in Rust using Location Graphs 137
5.6.1 An application of the owners-as-ombudsmen: a bank system 137
5.6.2 An application of the logger system: a Publish-Subscribe server 141

5.1 Programming model

In this section, we detail the choices we had to make in order to implement the location graph framework. For this implementation, we basically had the choice among two possibilities: write an ad-hoc location graph language; or integrate the location graph notions into an already existing language.

Of course, write a brand new language is appealing, since it allows one to design it to perfectly suit their needs. However, if one intend to write a useful language, they would need to reimplement a lot of unrelated stuff, from e.g. a basic parser to a full standard library, via a concurrent programming model. Without all those unrelated components, the created language would stay at the state of toy language, and would not show how the location graph framework could be used in practice.

On the other hand, by integrating the location graph framework into an existing language, all the unrelated elements come for free, but one has to find a way to tune the framework so it can be interfaced with the host language.

We chose the second approach, in which we integrate the location graph as a Rust library. This library offers structures and functions that anyone can use to program with a location graph approach in mind.

The first subsection is a general overview in which we explain the design choices we made to implement the various components of the framework. The second subsection shows the limitations of the implementation with respect to the theoretical framework.

5.1.1 Design choices

An object for locations. In the study of the theoretical framework, we considered locations as a whole. In practice, such approach is not practical: we have to let the user program the locations, while managing all the underlying mechanism. Therefore, some parts of the locations have to be provided by the user, and some other by us; which highlights the fact that we can not consider locations as a unique entity.

The design choice we made is to provide a *location handle* which provides a bridge between user code and the underlying location system. More precisely, we have a structure that provides primitives the user can call to, e.g. , change its sort, perform a transition, etc..

Roles. In the theoretical framework, roles are simply identifiers that can be bound by locations, and which implicitly create a communication medium between locations that bind the same identifier. In our implementation, we therefore have to take those two aspects (identifier and communication medium) into account.

On the first hand, on the implementation of the role identifiers, the main point is that we can not provide a unique implementation, for some systems may rely on a precise structure on role identifiers (e.g. some systems may require that identifiers are integers, some other that they are strings, etc..). Therefore, we only provide a *trait*[1] the user shall implement.

Communication channels, such as in the theoretical framework, are not directly accessible to the user. Instead, the location system stores used identifier, and each time a location binds a new identifier, a communication channel from the standard library (`std::sync::mpsc`) is instantiated, associated with that new identifier, and used for communications on that role.

Transitions. The theoretical framework does not specify how transitions are built: it only assumes that there is a set of transitions, and that, without specifying how, locations can take transitions. Of course, we need to adapt and refine this for an actual implementation.

First, while there is an infinite number of possible transitions, all can be described as the combination of a few atomic elements, such as, e.g., binding a role or creating a location. In our library, transitions are implemented with this approach. Locations can incrementally build up transitions (via some primitive functions), which are then fired.

In the theoretical framework, if no transition can be taken by a location, the location is locked. In practice, this is not suitable, as the user should (at least) get some feedback that no progress is possible. Therefore, we adopt a strategy similar to transactions in transactional memory models: when a location attempt to take a transition, there is no guarantee of success, but the transition is atomic (i.e. if it fails, then the state of the location does not changes, except for the feedback elements).

[1] An interface, in the Rust jargon.

Locations are threads. In our library, we assume each location runs on an independent thread. Rust offers multiple ways to implement concurrent systems, e.g. using lightweight threads or using async functions. Of course, we make the assumption that the underlying scheduler is fair, in the sense that no thread suffers starvation.

Among the four elements of locations (the process, the sort, the provided and required roles), we mentioned above that the location handle stores the sort and the provided and required roles. The process, on the other hand, is the function that is run on the thread (and which has access, in principle — but not necessarily, to a location handler structure).

> The current implementation internally uses `std::thread::yield_now` at some places, i.e. the assumed model of concurrency is the `std::thread` library.
> Of course, ideally, we would like to decouple as much as possible the location graph and the concurrency aspects of the system. This is absolutely possible in theory: the call to `yield_now` is simply here to improve performance, and all other concurrent aspects are directly managed by the user (e.g. the creation of new threads to instantiate new locations).
> We could absolutely define a trait to have a generic method to yield processes, and provide a default implementation the user could use or redefine. Nevertheless, this would imply an increased verbosity of (almost) all types used to take into account this new parameter, and I am not sure such trade-off is worth. I am curious to see some real use cases from third parties to make this choice.

Skeleton locations and skeleton graphs. While the location graph is implicit (there is no structure that holds the whole state of the location graph, it is implicitly formed by the different location threads and the bound roles), the skeleton location has a dedicated structure, shared among locations, that is transparently updated to reflect the current state of the graph. Of course, locations do not have access to that skeleton graph structure. Instead, only an immutable reference to that structure is provided as an argument of the authorisation function.

Skeleton graphs (and skeleton locations) are black boxes to the user. Instead, they can manipulate those elements via iterators and accessors. For instance, the programmer can filter skeleton locations of a skeleton graph to find all skeleton locations that have a certain sort.

Unconstrained location transition set and authorisation function. In the previous paragraph, we explained that attempting a transition can return an error. According to the theoretical framework, transitions must belong to the set of unconstrained location transitions of the instance, and they should satisfy the authorisation function, at the time the transition is taken. Therefore, we should give the user a way to model those two concepts.

While those two elements are different in the theoretical framework (one is a set, the other is a predicate), we implement them similarly. Since they must be specified by the user, and since the very base elements of a given instance are not known, we can not provide an implementation. Instead, we define two traits (one for the set of unconstrained location transitions, the other for the authorisation function), which both require a single function: for the set of unconstrained location transitions, the function indicates whether a given transition belongs to that set, and for the authorisation function, given a skeleton graph, a location and a transition, whether it is allowed.

Said otherwise, the user defines the set of unconstrained location transitions thanks to a predicate which states which transitions belongs to that state.

Remark. In the theoretical framework, the authorisation is stateless: it is a predicate over an environment and an unconstrained location transition.

In the implementation, having stateful authorisation transition came for free, we therefore adopt this possibility. ◁

Memory and sharing. The location graph framework assumes that there are no communication channel, of any kind, between locations, except for roles. We therefore have to enforce this isolation in order to correctly use the library.

This is a point where the choice of the host language helps: for instance, consider Java and the problem of shared memory (i.e. object aliases). Java does not provide any guarantee about references to objects, and would either require the programmer to be very careful about their memory management, or it would require us to take measures to detect and forbid unwanted aliases[2].

[2]In fact, an early implementation of the location graph framework was done in Java, and I can testify that this is a nightmare.

Kind of error	Policy implemented
Not in the unconstrained location transition set	Error reported
Authorisation function not satisfied	Error reported
Message exchanged on unbound role	Deadlocks
Failed expected value	Deadlocks
Binding error	Panic

Figure 5.1: Transition failure policies.

On the other hand, the Rust programming language provides, thanks to its typesystem, a mechanism for alias control (aliases are not possible, except when explicitly declared). Therefore, with the choice of Rust, and a programming discipline easy to ensure[3], we can enforce the required isolation between locations.

Remark. We require *the user* not to share memory. However, the code of the library uses, at some places, some shared constructs (e.g. a unique shared skeleton graph). This usage is scarce, controlled, and it correspond to some shared structures of the abstract machine we will develop below. Therefore, we assume the shared memory used internally is correct (i.e. it does not leak information). ◁

In case of failure. In the theoretical framework, if no transition is possible, the instance is locked. How does this behaviour translate in the actual implementation?

Three approaches are possible, from the most to the least suitable, from the user perspective:

— When committing a transition, the function may return a success or failure value (using usual `Error` types). In case of failure, nothing happened for the location (i.e. its state is the same than before the (failed) commit), whatever *state* means. This allows recovery.
— The system deadlocks.
— The system panics.

Also, depending on the kind of failure, we may adopt a different policy: e.g. we may not necessary expect the same behaviour if we try to send a value on a role we did not bind beforehand than when the authorisation function fails.

Typically, if the authorisation function fails, we expect, at least, to lock the system, if not just rejecting the transition with a causal explanation of the rejection. The failure policies, as of September 25$^{\text{th}}$, 2020, are shown in Figure 5.1.

5.1.2 Divergences from the theoretical framework

In the discussion above, we presented the main design choices for our implementation, and how we managed to stay close to the theoretical framework. For the sake of completeness, however, we have to mention a few aspects in which the implementation differs from the theory.

One can distinguish various reasons for the existence of those differences. First, the theoretical model might not be suitable for practical use (see e.g. the following paragraph on early and late binding); second, the language chosen for the implementation (Rust) has its semantics and we have to deal with it (see e.g. the paragraph on the constraints on the basic types); finally, the implementation development was constrained by time and some aspects of the library were less a priority than others, with respects to the objectives and the originality of the work (e.g. the discussion above about failure; the notion of failure is not present in the theoretical work, and we have to introduce it, yet, we chose, for the sake of time saving, to panic in cases where it is possible, with more time, to simply report the error).

Fundamentally, the differences are not that important, and we claim that they are reasonable with respect to our goals. However, it might be important, for a future work of stronger formalisation (e.g. for a Coq implementation) to consider what can be done to reduce the gap between theory and implementation. On this subject, my opinion is that: (i) it would be interesting to have a study of the model using late binding semantics[4]; (ii) the question of priorities is probably the hardest one, as they are essential for

[3] Since aliasing must be explicit, it is much less likely to *accidentally* share memory.

[4] Whether switching from early binding to late binding semantics causes problems on the already existing theory is a good question. I can not think of a fundamental problem that would appear by this change *in the work presented in this thesis*. However, I do not know for the other results.

some key aspects of the theory developed[5], yet it is really hard to think of a good way to implement them; (iii) differences induced by the underlying programming language are, in my opinion, less important: one should take care of aliases (since our work is, fundamentally, to prevent unwanted aliases), and a language like Rust really helps with that[6]; except the question of aliases, the language mostly imposes additional constraints, meaning it does not affect the safety of the library, but it (potentially) prevents the possibility to write some instances, or imposes a discipline; and (iv) finally, differences due to the lack of time are more annoying than really dangerous: they prevent to have some comfort when writing applications with our framework; but, in the meantime, they are problems that are known and that could be easily tackled provided enough time.

Early binding and late binding. Strictly speaking, the semantics of the theoretical framework is an early binding semantics, that is, the receiver should *guess* the value it will receive, and if the guess is correct, then both locations synchronise and the transition is taken.

This is appealing since it allows transitions of the kind *if I receive value V on role r, then do …* (which we call the *expect* kind below), and *receive value X on role r* (called the *receive* kind hereafter). In particular, those two kinds of transitions can be performed during the *same* transition: think of set of unconstrained location transitions $\Delta \cdot \emptyset \rhd L \xrightarrow{\Lambda} L'$, with $\Lambda.\texttt{sync} = \{\hat{r} : \overline{a}\langle V\rangle, \hat{r} : \overline{a}\langle x\rangle\}$ for any possible value of x.

This is possible in theory because we can allow the receiver to take a transition for any possible value for x, and the semantics performs the *selection* of which value is indeed received.

In practice, this would be much less convenient: the location would have to propose a transition for each possible value, and the underlying system would have to decide which one should be taken, depending on the other locations. While we have the possibility to propose multiple transitions at once, the complexity explodes with respect to the number of proposed transition, therefore taking the theoretical approach is not suitable in practice. Also, in practice, the user *knows* whether they want a *expects* or *receives* kind of action.

Therefore, we chose the following approach, which allows both kinds of messages.

— We assume we can determine which messages correspond to an *expects* and which correspond to a *receive* action. More specifically, we have three primitive to exchange messages: *send*, *receive* and *expect*. The *send* and *expect* primitives require a message. A *send* primitive matches a *require* primitive (and the message is exchanged). An *expect* primitive is matched by an other *expect* on the other side of the role, if both expect the same value.

— We reduce the complexity of finding a possible matching by enforcing an order relation on *expected* values.

Constraints on messages. The theoretical framework does not consider practical constraints on value exchanged: the set of values is given. When implementing the framework, some questions arise. It is quite natural to have a data type for messages, and to let the user define this type. Therefore, in the library, the type of messages is generic (generic type `M` in our implementation).

The goal is to impose as few constraints on `M` as possible, such that the user has a lot of freedom in the messages it can send. These constraints are expressed as which traits must be implemented by the type of messages.

In the library, we therefore make the following choices, while trying to stick to Rust standard communication protocols. Rust channels[7] requires messages to implement the `Send` trait[8].

Therefore, we also require messages to have the `Send` marker.

Concerning *expect* messages, we need to be able to compare to messages. Therefore, these messages need to implement the `Eq` trait.

This choice implies some limitations. For most use cases, the constraints are not problematic, as most types we think of at first implement the required traits. However, some types do not. Typically, `std::io::Stdin` does not implement `Clone`.

While it could be possible not to require that exchanged messages implement `Eq`, and that expected messages implement `Send`, we chose to unify both kinds of messages in a single trait (i.e. to put unnecessary

[5] Once again, in this work, we did not use priorities that much, but they are mandatory for other results.

[6] Actually, a first implementation was attempted in Java, which basically lead to rewriting all objects in order to keep track of aliases and caused a much more complex library, even though it was much less advanced than the one presented here.

[7] See the `std::sync::mpsc` module.

[8] Strictly speaking, `Send` is a marker. As the name suggest, markers use the trait mechanism to expose some properties, here the fact that the data can be safely send. Markers traits do not require to implement any methods, and they are inferred by the compiler.

constraints on messages), in order to simplify the framework.

5.2 An abstract machine for location graphs

Before presenting the implementation of the library, we describe an abstract machine, which is an intermediate step between the theoretical framework and the actual implementation presented below.

This abstract machine is introduced in two steps: first, we focus on a single location: we describe how we implement locations and what we call the *local* semantics, i.e. the semantics of that locations, ignoring all surrounding elements. Of course, such semantic is highly non-deterministic, for instance when receiving messages: the value of the message is not specified.

In a second step, we describe location graphs and their *global* semantics. As in the theoretical framework, location graphs are just a composition of locations. The global semantics is defined, at first approximation, as a concurrent composition of local transitions.

In Section 5.2.2 we describe the local behaviour; then, in Section 5.2.3 we explain the global aspects of the abstract machine.

5.2.1 Preliminary definitions

Sets. Let (i) $\mathbb{R}$ be the set of role identifiers; (ii) $\mathbb{D} = \{\texttt{provided}, \texttt{required}\}$ the set of role directions; (iii) $\mathbb{M}$ the set of messages; and (iv) $\mathbb{S}$ the set of sorts. We assume $\mathbb{R}$, $\mathbb{M}$, and $\mathbb{S}$ are provided (by the user).

Endpoints. We will see below that, in order to get closer to the actual implementation, locations in the abstract machine have a single set of *endpoints*, instead of the separate set of provided and required roles presented in the theoretical part of this work. Endpoints are basically a role name with a direction:

Definition 86 (Endpoint)**.** Endpoints are elements of the set $\mathbb{E} = \mathbb{R} \times \mathbb{D}$.

Skeleton locations and skeleton graphs. A skeleton location is a tuple containing a sort and a set of endpoints.

Definition 87 (Skeleton locations (Abstract machine))**.** The set of skeleton locations is:

$$\mathbb{\Sigma}_l \triangleq \mathbb{S} \times \mathcal{P}(\mathbb{E})$$

A skeleton graph is a multiset of such skeleton locations[9].

Definition 88 (Skeleton location graphs (Abstract machine))**.** The set of skeleton graphs is:

$$\mathbb{\Sigma} \triangleq \mathcal{P}^\star(\mathbb{\Sigma}_l)$$

5.2.2 Transitions, locations and local semantics

The goal of this first subsection is to explore the behaviour of a single location. Writing an abstract machine presents new challenges we did not face in the theoretical part. The main one is the nature of transitions: in the theoretical framework, we simply assume transitions existed, and where defined thank to the unconstrained location transition. Now, we have to actually define a structure which represent the changes of a transition, so that our location structure can instantiate such structure and provide it to the underlying system.

Once we have this transition structure, we define our locations. We will base our semantics on the notion of *primitive functions*, which are to be used by the programmer to change the internal state of our locations, and therefore form an interface between the user space and the underlying system. In that section, we therefore define those primitives.

The third part of this section is the definition of the local semantics, based on the elements described previously, which we illustrate in depth with an example.

Through this section, we also define some helper functions that will be needed latter, related to the elements mentioned above.

[9] It is possible to have multiple occurrence of locations that have no bound roles.

Transition items and transitions. Our goal is to define a structure the user can manipulate to represent transitions, which they can then provide to the system in order to fire the represented transition. We choose to take an incremental approach where the transition is built in steps from basic blocks which we call *transition items*, and which represent atomic[10] changes that can be performed on a location during a transition. Notice that transition items are not functions, but records of elements.

Transition items are elements of:

Definition 89 (Transition item)**.**

$$\texttt{Bind}(sk, r, d) \qquad \texttt{Create}(s) \qquad \texttt{Expect}(r, m, d) \qquad \texttt{Receive}(r, d) \qquad \texttt{Release}(sk, r, d)$$

$$\texttt{Send}(r, m, d) \qquad \texttt{Sort}(sk, s) \qquad \texttt{Remove}(sk)$$

where $r \in \mathbb{R}$, $m \in \mathbb{M}$, $d \in \mathbb{D}$, $s \in \mathbb{S}$ and $sk \in \mathbb{\Sigma}_l$. The set of transition items is noted $\mathbb{T}_i$.

The intuition behind each of those items is the following: (i) $\texttt{Bind}(sk, r, d)$ binds a role r in the direction d; (ii) $\texttt{Create}(s)$ instantiates a new location which initial sort is s; (iii) $\texttt{Expect}(r, m, d)$ attempts to synchronise on role r (bound in direction d) with the value m; (iv) $\texttt{Receive}(r, d)$ receives a value on role r (bound in direction d); (v) $\texttt{Release}(sk, r, d)$ releases the role r which is currently bound in direction d; (vi) $\texttt{Send}(r, m, d)$ sends a message m on the role r (bound in direction d); (vii) $\texttt{Sort}(sk, s)$ changes the sort to s; and (viii) $\texttt{Remove}(sk)$ removes the current location. In each of those items, sk is the skeleton of the current location, and serves an administrative purpose in the underlying machinery.

Then, it becomes easy to define a transition: it is a multiset of transition items[11].

Definition 90 (Transition (Abstract machine))**.** Let $\mathbb{T}$ be the set of transitions:

$$\mathbb{T} \triangleq \mathcal{P}^\star(\mathbb{T}_i)$$

Also, we define the operation $\oplus$ which merges two transitions and removes matching transition items. It is typically used to collect unmatched transition items.

Definition 91 (Transition item sum)**.** Given two transition items t_i and t_i', $t_i \oplus t_i'$ is defined as the least commutative operation such that:

$$t_i \oplus t_i' \triangleq \begin{cases} \emptyset & \text{if } t_i = \texttt{Expect}(r, v, d) \wedge t_i' = \texttt{Expect}(r, v, d') \wedge d \neq d' \\ \emptyset & \text{if } t_i = \texttt{Receive}(r, d) \wedge t_i' = \texttt{Send}(r, m, d') \wedge d \neq d' \\ \emptyset & \text{if } t_i = \texttt{Bind}(sk_i, r, d) \wedge t_i' = \texttt{Release}(sk_i', r, d) \\ \{\!\{t_i, t_i'\}\!\} & \text{otherwise} \end{cases}$$

Notice that the operation returns a multiset, since it is possible that $t_i = t_i'$, typically if a message is sent multiple times; and we want to keep track of those multiple items.

We can now define our sum of full transition using individual item sums:

Definition 92 (Transition sum)**.** Given two transitions t_1 and t_2, $t_1 \oplus t_2$ is defined as:

$$t_1 \oplus t_2 \triangleq \begin{cases} t_1' \oplus t_2' & \text{if } \exists t_i^1, t_i^2 \cdot t_1 = \{\!\{t_i^1\}\!\} \cup t_1' \wedge t_2 = \{\!\{t_i^2\}\!\} \cup t_2' \wedge t_i^1 \oplus t_i^2 = \emptyset \\ t_1 \cup t_2 & \text{otherwise} \end{cases}$$

Naturally, $\oplus$ is associative, commutative and $\emptyset$ is a neutral element[12].

Notation. We note $\overset{\oplus}{\sum}_{t \in \{t_1, \ldots, t_n\}} t$ for $t_1 \oplus \ldots \oplus t_n$.

Example 9 (Transition sum)**.** Consider the two transitions

$$t_1 = \{\!\{\texttt{Receive}(r_1, \texttt{provided}), \texttt{Sort}(sk_1, s_1)\}\!\}$$

[10] Note that here, we use *atomic* in its original sense: *which can not be split*; and not in its concurrency theory meaning.
[11] It is possible to have multiple occurrence of the same transition item, e.g. to send multiple times the same message.
[12] It is an addition on transitions, hence the symbol $\oplus$.

and

$$t_2 = \{\!\{\texttt{Send}(r_1, m, \texttt{required}), \texttt{Create}(s_2)\}\!\}$$

We have:

$$t_1 \oplus t_2 = \{\!\{\texttt{Sort}(sk_1, s_1), \texttt{Create}(s_2)\}\!\}$$

An other helper function for transitions is $\text{skl}_i(t_i)$. It is used to retrieve the skeleton locations embedded in transition items.

Notice it returns a set of skeleton location. Indeed, for transition items that don't have local effects on the skeleton graph, there is no need to identify to which skeleton location it applies.

Definition 93 (Skeleton of a transition item)**.**

$$\text{skl}_i(t) = \begin{cases} \{skl\} & \text{if } t \in \{\texttt{Bind}(skl, \_, \_), \texttt{Release}(skl, \_, \_), \texttt{Sort}(skl, \_)\} \\ \emptyset & \text{otherwise} \end{cases}$$

We extend this function from transition items to transitions:

Definition 94 (Skeleton of a transition)**.**

$$\text{skl}(t) = \bigcup_{t_i \in t} \text{skl}_i(t_i)$$

Remark. For a transition t, if $\text{skl}(t)$ contains more than one element, it means that the transition contains items that apply to multiple skeleton location.

This kind of transition are avoided: in practice, a transition is emitted by a single location and all skeleton location embedded in transition items are the skeleton of the current location, and are therefore all equal.

However, it is possible that $\text{skl}(t)$ is empty: this means that t has no local effect on the skeleton graph. For instance, this happen if t contains $\texttt{Create}(\_)$ or $\texttt{Remove}(\_)$[13] (which have global effects), or $\texttt{Send}(\_, \_, \_)$, $\texttt{Receive}(\_, \_)$ or $\texttt{Expect}(\_, \_, \_)$ (which have no effects on the skeleton graph). ◁

In order to characterize our intuitive notion of transition, we define the following predicate, which defines what we call *well-formed* transitions. We remove unsound transitions, such as changing twice the sort or binding and releasing in the same transition.

Definition 95 (Well-formed transition (Abstract machine))**.** We say a transition is well formed if and only if the following predicate holds:

$$\begin{aligned} \texttt{WF}_t(t) \Leftrightarrow & \forall sk, r, d \cdot \neg(\texttt{Bind}(sk, r, d) \in t \wedge \texttt{Release}(sk, r, d) \in t) \\ & \wedge \forall s_1, s_2 \cdot \texttt{Sort}(\_, s_1) \in t \wedge \texttt{Sort}(\_, s_2) \in t \Rightarrow s_1 = s_2 \\ & \wedge |\text{skl}(t)| \leq 1 \end{aligned}$$

We now specify the possible results of a given transition. A transition can either succeed or fail. In case of success, a set of side effects is returned. In case of failure, an error is returned which can either be: (i) a *not allowed* error, meaning that the transition is not allowed in the current policy; (ii) a *not in transition set* error, meaning that the transition is not in the set of unconstrained location transitions; (iii) a *not selected* alert, meaning that the transition is valid, but an other valid transition was selected instead; and (iv) a *no match* error, meaning that there was an error in matching some expected values.

Remark. Detecting *no match* errors is not covered in this thesis. This does not break safety: if there is a matching error, the concerned transition simply hang indefinitively, deadlocking the locations.

We still introduce the error so that we can, in the future, simply plug the detection mechanism, and location programs are suppose to manage this error (even tho it can not occur in the current implementation). ◁

[13] One could argue that $\texttt{Remove}(\_)$ is a local effect, or at least should be reflected in *skl*. The fact is that we treat it like the location creation, and we don't need to gather this information.

A success simply indicates which values are received on which roles. Finally, results are either a failure or a success.

Definition 96 (Transition result)**.** An error is an element of the set $\mathbb{E}_r \triangleq \{\texttt{Not\_In\_Trans\_Set}, \texttt{Not\_Selected}, \texttt{Not\_allowed}, \texttt{No\_match}\}$.
A success is an element of the set $\mathbb{S}_r \triangleq \mathcal{P}^\star(\{\!\{\langle m, r\rangle | m \in \mathbb{M} \wedge r \in \mathbb{R}\}\!\})$
A result is an element of $\mathbb{R}_t \triangleq \mathbb{E}_r \cup \mathbb{S}_r$.

Remark. Note that $\mathbb{E}_r$ and $\mathbb{S}_r$ are disjoints. ◁

Notation. We note $r = \lfloor s \rfloor$ if r is a success s, and $r = \lceil e \rceil$ if r is an error e.

Notice that the set $\mathbb{R}_t$ defines all possible results of all transitions. To get the set of possible results of a given transition, we define the helper function $\text{msg}(t)$ which returns the set of all possible multisets of messages received (associated with the role the message is received on) when firing t. The definition may seem a bit odd, but is easily explained: $\text{msg}(t)$ returns a set of multiset of messages received. Therefore, if t contains a $\texttt{Receive}(r, \_)$ item (i.e. a successful firing of t receives a message on r) and we call t' the remaining of t, then one can receive any value on r: any $\langle r, m\rangle$ can be received, for any $m \in \mathbb{M}$. On the other hand, $\text{msg}(t')$ returns, by recursivity, the set of all multisets of messages received in the remaining of t. Therefore, to get the all possible multisets of messages received in t, we simply have to add each possible $\langle r, m\rangle$ to each possible m' of $\text{msg}(t')$[14].

Definition 97 (Messages of a transition)**.**

$$\text{msg}(t) = \begin{cases} \{\!\{\langle r, m\rangle\}\!\} +^\star m' | m \in \mathbb{M} \wedge m' \in \text{msg}(t')\} & \text{if } t = \texttt{Receive}(r, \_) \cup t' \\ \emptyset & \text{otherwise} \end{cases}$$

Similarly, we want to retrieve the effect of a transition t on the sort of a location. We define $\text{sort}(s, t)$ which returns a new sort modified by t if any, or the default s otherwise. Fortunately, it is much easier to define than $\text{msg}(t)$:

Definition 98 (New sort in a transition)**.**

$$\text{sort}(s, t) = \begin{cases} s' & \text{if } \texttt{Sort}(\_, s') \in t \\ s & \text{otherwise} \end{cases}$$

Remark. Notice this is not a function: without further constraints on the transition, $\text{sort}(s, t)$ may relate to multiple sorts, e.g. if the transition contains different $\texttt{Sort}(\_, s)$ items.
In our case, we will always consider well-formed transitions, in which case $\text{sort}(s, t)$ is a function. ◁

Finally, we also want to track the changes in the endpoints that are done when a transition t is fired. The relation $\text{endpoints}(e, t)$ returns the effect of t on the initial set of endpoints e:

Definition 99 (Endpoints effects of a transition)**.**

$$\text{endpoints}(e, t) = \begin{cases} \text{endpoints}(e \cup \{\langle r, d\rangle\}, t') & \text{if } t = \{\texttt{Bind}(\_, r, d)\} \cup t' \\ \text{endpoints}(e \backslash \{\langle r, d\rangle\}, t') & \text{if } t = \{\texttt{Release}(sk, r, d)\} \cup t' \\ e & \text{otherwise} \end{cases}$$

Remark. This is not a function too because the order in which endpoints are added or removed matters: consider $\text{endpoints}(\emptyset, \{\!\{\texttt{Bind}(sk, r, d), \texttt{Release}(sk, r, d)\}\!\})$; if $\texttt{Bind}(sk, r, d)$ is evaluated first, it overall evaluates to $\emptyset$; however, if $\texttt{Release}(sk, r, d)$ is evaluated first, the overall evaluation is $\{\langle r, d\rangle\}$.
Again, if the transition t is well formed, this can not happen and $\text{endpoints}(e, t)$ is a function. ◁

[14] This is actually easier to understand if one think of (nested) tuples instead of multisets: at each step, we would do the cross product of $\{\langle r, m\rangle | m \in \mathbb{M}\}$ and $\text{msg}(t')$. We use multiset to flatten the tuples and because order is not relevant.

Location primitives, location handles and locations. We now have all the infrastructure to define locations of the abstract machine. We begin by defining the primitives we will be using. Then we continue with the location handles, i.e. the structure the user manipulates to interact with underlying system. Finally, we define the locations *per se*.

We intend to build locations with a semantics that allows multiple transitions to be proposed at once by a single location, and where a single one of those transitions is selected and fired.

To define the behaviour of locations, we abstract away the details of location programs[15]. We only keep a minimal set of primitives that programs use to interact with the location system. Primitives are the basic building block used to set up a transition and fire it. Most of the primitives are used to build incrementaly a transition. Two additionnal primitives are provided to control the set of primitives of a location: one to reset that set, and the other to add a new transition. Notice that we have a notion of *transition being built*: our primitives that add new transition items modify the last transition added to the set, the previous one being considered finished.

Definition 100 (Primitives of the abstract machine)**.**

$$\mathit{silent} \quad \mathit{bind}(\mathit{rid}, \mathit{dir}) \quad \mathit{release}(\mathit{rid}) \quad \mathit{create}(\mathit{sort}) \quad \mathit{newSort}(s) \quad \mathit{receive}(\mathit{rid})$$

$$\mathit{send}(\mathit{rid}, \mathit{addr}) \quad \mathit{expect}(\mathit{rid}, \mathit{addr}) \quad \mathit{reset} \quad \mathit{next\_transition} \quad \mathit{end} \quad \mathit{commit}$$

The set of primitives is the set $\mathbb{P}$.

Intuitively, those primitives do the following: (i) the primitive *silent* denotes an internal computation that has no effect on the elements of the location (typically, some arbitrary computation); (ii) the primitives $\mathit{bind}(\mathit{rid}, \mathit{dir})$, $\mathit{release}(\mathit{rid})$, $\mathit{create}(\mathit{sort})$, $\mathit{newSort}(s)$, $\mathit{receive}(\mathit{rid})$, $\mathit{send}(\mathit{rid}, \mathit{addr})$ and $\mathit{expect}(\mathit{rid}, \mathit{addr})$ are primitives used to build up transition, each of them enqueues the corresponding transition item to the transition being built; (iii) the primitive *reset* is used to reset the set of transitions that are currently being built; (iv) by calling the primitive *next_transition*, the user indicates it terminates the current transition and begins a new one (note that the transition that terminates is not committed: this primitive is there to allow the location to propose multiple transitions at once); (v) the primitive *end* terminates the current location (it inserts a `Remove`(sk) transition item); and (vi) the primitive *commit* commits all transitions, i.e. it let the underlying system select and fire one of the built transitions.

Similarly, we abstract away memory details. We suppose programs have a state σ, and we define some functions to manipulate the states. Let $\mathbb{S}_t$ be the set of states. We suppose there exists a special initial state $\sigma_i \in \mathbb{S}_t$, and we are given a function $\text{memory}(\cdot) : \mathbb{S}_t \to \mathbb{A} \twoheadrightarrow \mathbb{M}$, which returns a map from adresses to objects, where addresses are taken from a set $\mathbb{A}$ which include a special adress `ret` used for returned values.

Notation. In practice, and to stay closer to the usual definition of memory, we use the usual *squared bracket* notation used for maps: if $m_\sigma = \text{memory}(\sigma)$, then $m_\sigma[a]$ denotes $m_\sigma(a)$.

Notation (Memory update). Let $m_\sigma = \text{memory}(\sigma)$ be the memory of a state σ.

$$m[a \mapsto o](a') \triangleq \begin{cases} o & \text{if } a = a' \\ m(a') & \text{otherwise} \end{cases}$$

$$m[\times a](a') \triangleq m(a') \text{ if } a \neq a'$$

In order to stay close to the implementation, we distinguish locations and location handles (See Section 5.3.2).

Locations are similar to actors in an actor system: it is a thread of computation, with a private stack, which interacts, via some primitives, with the rest of the system.

To interact with the system, each location has a *location handle*. A location handle is a placeholder which stores the data required to perform the transitions (the sort, the transitions being built, the roles, and the result of the last transition committed).

Definition 101 (Location handle)**.** A location handle is an element of $\mathbb{L}_H \triangleq \mathbb{S} \times \mathcal{P}^+(\mathbb{T}) \times \mathcal{P}(\mathbb{E}) \times \mathbb{R}_t$.

[15] Typically, in our case, the program of a location is a Rust program.

The set of transitions of a location handle is well formed with respect to that handle if (i) each transition in the set is a well formed transition; and (ii) for each transition, for each item of that transition that embeds a skeleton location, the skeleton location is the skeleton of the given location.

Definition 102 (Well-formed handle)**.**

$$\begin{aligned}\texttt{WF}_h(\langle s, tl, e, r\rangle) \Leftrightarrow \forall t \in tl \cdot \forall \texttt{Bind}(sk, r, d) \in t \cdot sk = \langle s, e\rangle \\ \wedge \forall \texttt{Release}(sk, r, d) \in t \cdot sk = \langle s, e\rangle \\ \wedge \forall \texttt{Remove}(sk) \in t \cdot sk = \langle s, e\rangle \\ \wedge \texttt{WF}_t(t)\end{aligned}$$

Then, a location is simply the aggregate of a state and a location handle.

Definition 103 (Location (Abstract machine))**.** A location is an element of $\mathbb{L} \triangleq \mathbb{S}_t \times \mathbb{L}_H$.

Notice that the location handle of a location does not lies in its state. It illustrates the fact that the program *does not* have (a direct) access to the handle's fields. This is reflected below, in the local semantics, in which the handle is modified in a controlled way.

Notice also that we assume (i) that the state is private, i.e. there is no shared memory between locations, and (ii) that locations are isolated (there is no communication channel between two locations), except for the communication primitives provided by the location handle.

As we saw previously (Section 5.1.1), these two hypothesis are reasonable in the actual implementation, thanks to the very nature of Rust.

In addition, we define a function which enqueues a new transition item in the list of transitions (notice that this function creates a new transition if the current list is empty).

Definition 104 (Transition item enqueuing)**.**

$$\text{enqueue}(ts, t_i) \triangleq \begin{cases} [\{\!\{t_i\}\!\}] & \text{if } ts = [] \\ (t \cup \{\!\{t_i\}\!\}) :: tl & \text{otherwise, if } ts = t :: tl \end{cases}$$

Remark. With such a list, the primitive *next_transition* can be implemented naively: one just need to add a new $\emptyset$ at the head of the list. This is done in the (New Transition) rule below. ◁

Finally, we want to know the possible results a location can get, given a provided list of transitions.

Definition 105 (Possible results of a list of transitions)**.** Given $ts \in \mathcal{P}^+(\mathbb{T})$, $rs \in \mathcal{P}^+(\mathbb{R}_t)$ and $e \in \{\top, \bot\}$, the set of possible results of tl is:

$$\text{possible\_result}(ts, rs, e) \triangleq \begin{cases} true & \text{if } ts = [] \wedge rl = [] \wedge e = \bot \\ \text{possible\_result}(tl, rl, \bot) & \text{if } ts = hd\_t :: tl \wedge rs = hd\_r :: rl \\ & \quad \wedge\, hd\_r \in \text{msg}(hd\_t) \wedge e = \top \\ \text{possible\_result}(tl, rl, e) & \text{if } ts = hd\_t :: tl \wedge rs = hd\_r :: rl \wedge hd\_r \in \mathbb{E}_r \\ false & \text{otherwise} \end{cases}$$

Remark. The role of the argument e is to ensure that there is one and only one $\text{msg}(t)$ in the list of results; meaning that only one transition was selected, and that the other where not, due to the reason given in the error. ◁

Local semantics. To conclude this first subsection, we put all the above together, in order to define the *local semantics* of locations of the abstract machine.

First, we suppose we are given a partial function $\text{next}(\cdot) : \mathbb{L} \nrightarrow (\mathbb{P} \times \mathbb{S}_t)$ which returns the next primitive of the program with the updated program state, given the current state and the current elements of the location handle[16].

We begin by the rules used to build up a transition. Those rules relate primitives with the transition items.

[16] Intuitively, to define next(·), we can lookup elements from the location handle, but not modify them.

$$\text{(Silent)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{silent},\sigma'\rangle}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,t,e,r\rangle\rangle} \qquad \text{(Reset)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{reset},\sigma'\rangle}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,[],e,r\rangle\rangle}$$

$$\text{(New Transition)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{next\_transition},\sigma'\rangle}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\emptyset::t,e,r\rangle\rangle}$$

$$\text{(Send)}\ \frac{\begin{array}{c}\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{send}(r_{id},a),\sigma'\rangle \qquad \exists d\in\mathbb{D}\cdot\langle r_{id},d\rangle\in e\\ \text{memory}(\sigma')=\text{memory}(\sigma)[\times a] \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Send}(r_{id},\text{memory}(\sigma)[a],d)),e,r\rangle)\end{array}}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Send}(r_{id},\text{memory}(\sigma)[a],d)),e,r\rangle\rangle}$$

$$\text{(Expect)}\ \frac{\begin{array}{c}\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{expect}(r_{id},a),\sigma'\rangle \qquad \exists d\in\mathbb{D}\cdot\langle r_{id},d\rangle\in e\\ \text{memory}(\sigma')=\text{memory}(\sigma)[\times a] \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Expect}(r_{id},\text{memory}(\sigma)[a],d)),e,r\rangle)\end{array}}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Expect}(r_{id},\text{memory}(\sigma)[a],d)),e,r\rangle\rangle}$$

$$\text{(Sort)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{newSort}(s'),\sigma'\rangle \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Sort}(\langle s,e\rangle,s')),e,r\rangle)}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Sort}(\langle s,e\rangle,s')),e,r\rangle\rangle}$$

$$\text{(Bind)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{bind}(r_{id},\mathit{dir}),\sigma'\rangle \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Bind}(\langle s,e\rangle,r_{id},\mathit{dir})),e,r\rangle)}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Bind}(\langle s,e\rangle,r_{id},\mathit{dir})),e,r\rangle\rangle}$$

$$\text{(Release)}\ \frac{\begin{array}{c}\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{release}(r_{id}),\sigma'\rangle\\ \langle r_{id},d\rangle\in e \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Release}(\langle s,e\rangle,r_{id},d)),e,r\rangle)\end{array}}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Release}(\langle s,e\rangle,r_{id},d)),e,r\rangle\rangle}$$

$$\text{(Create)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{create}(ns),\sigma'\rangle \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Create}(ns)),e,r\rangle)}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Create}(ns)),e,r\rangle\rangle}$$

$$\text{(End)}\ \frac{\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{end},\sigma'\rangle \qquad \mathtt{WF}_h(\langle s,\text{enqueue}(t,\mathtt{Remove}(\langle s,e\rangle)),e,r\rangle)}{\langle\sigma,\langle s,t,e,r\rangle\rangle\to\langle\sigma',\langle s,\text{enqueue}(t,\mathtt{Remove}(\langle s,e\rangle)),e,r\rangle\rangle}$$

Remark. The rule (Silent) is very liberal and allows almost any transition. We let the user define next([·]) properly to have a smaller relation adequate to their needs. ◁

Remark. Even in case of (Expect), the message is removed from the stack. If the user wants to keep the value, it should be copied beforehand. ◁

Finally, when a transition is ready, it is *committed*. This is triggered using the primitive *commit*

$$\text{(Commit)}\ \frac{\begin{array}{c}\text{next}(\langle\sigma,\langle s,t,e,r\rangle\rangle)=\langle \mathit{commit},\sigma'\rangle\\ \text{possible\_result}(t,r',\top) \qquad s'=\text{sort}(s,t) \qquad e'=\text{endpoints}(e,t)\end{array}}{\langle\sigma,\langle s,t,e,r\rangle\rangle\xrightarrow{t}\langle\sigma',\langle s',\emptyset,e',r'\rangle\rangle}$$

The semantics of locations shows that transitions are built incrementally (with rules (Send), (Expect), (Sort), (Sort), (Bind), (Release), (Create) and (End)). When a list of transitions is ready, the location can take a commit its transitions with rule (Commit). At the local level, all commit transitions can be taken (the actual result is not specified). The effective choice is constrained by the global semantics (see below). Notice the rules (Silent) and (Reset) which are used to model local modifications of the state and the reset of the pending transition.

Finally, notice that the (Commit) rule only describes local modifications of the location: creation of new locations or the deletion of the location are not taken into account in the rule. The function $\text{apply}_l(\cdot,\cdot,\cdot)$, defined and used below (Definition 116 and in the global semantics), describes the full effects of a transition.

Example 10 (Synchronous integer counters)**.** We want to implement a system with two locations that synchronously count integers. For that, our state includes an integer $n\in\mathbb{N}$.
To synchronously count, each location successively perform transitions in which it both receives the current value and send the next one[a].

Therefore, for each location, we will: (i) add a (Send) transition item, with an incremented copy of the local counter; (ii) add a (Receive) transition item; (iii) commit the transition; and (iv) update the local counter in the state and loop. This procedure is easily implemented using a small automaton with 4 states. For convenience, our set of (automaton) states is: $\mathcal{S} = \{(\text{i}), (\text{ii}), (\text{iii}), (\text{iv})\}$. In addition, we identify each location with an integer taken in $\{0, 1\}$. Therefore, our set of location states is $\mathbb{S}_t = \{0, 1\} \times \mathbb{N} \times \mathcal{S}$.
We will have two roles: $\mathbb{R} = \{r_0, r_1\}$. Our messages are integers ($\mathbb{M} = \mathbb{N}$). We do not use sorts, so we use the trivial $\mathbb{S} = \{\top\}$ set.
Our set of addresses contains a single address: $\mathbb{A} = \{a\}$, and our memory function is simply returns our incremented counter: $\text{memory}(\langle n, s\rangle) = a \mapsto (n+1)$ if $s = (\text{i})$. Notice that our memory function has to comply with the fact that the value is not accessible after sending it, hence the condition.
Finally, our $\text{next}(\cdot)$ function is defined as follow:

$$\text{next}(\langle\langle i, n, s\rangle, \langle s, t, e, r\rangle\rangle) \triangleq \begin{cases} \langle \mathit{send}(r_i, a), \langle i, n, (\text{ii})\rangle\rangle & \text{if } s = (\text{i}) \\ \langle \mathit{receive}(r_{1-i}), \langle i, n, (\text{iii})\rangle\rangle & \text{if } s = (\text{ii}) \\ \langle \mathit{commit}, \langle i, n, (\text{iv})\rangle\rangle & \text{if } s = (\text{iii}) \\ \langle \mathit{silent}, \langle i, n+1, (\text{i})\rangle\rangle & \text{if } s = (\text{iv}) \text{ and } r = [\lfloor \{\!\{\langle r_{1-i}, n+1\rangle\}\!\} \rfloor] \end{cases}$$

For the sake of simplicity, we assume the roles r_0 and r_1 are properly bound initially, therefore we do not need to add the initialisation in the automaton.
Using our local semantics, such a location could run as follow:

$\langle\langle 0, 1, (\text{i})\rangle, \langle \top, [], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, \_\rangle\rangle$
$\rightarrow$ (Send)
$\langle\langle 0, 1, (\text{ii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided})\}\!\}], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, \_\rangle\rangle$
$\rightarrow$ (Receive)
$\langle\langle 0, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, \_\rangle\rangle$
$\rightarrow$ (Commit Success)
$\langle\langle 0, 1, (\text{iv})\rangle, \langle \top, [], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, [\lfloor \{\!\{\langle r_1, 2\rangle\}\!\} \rfloor]\rangle\rangle$
$\rightarrow$ (Silent)
$\langle\langle 0, 2, (\text{i})\rangle, \langle \top, [], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, \_\rangle\rangle$
...

Notice that, when performing the (Commit Success) rule, we assume the value received is 2. If it is something else, or a failure, the location deadlocks.

[a] This is, of course, not the smartest way to implement such a counter, but we take this approach to illustrate how messages are exchanged.

5.2.3 Graphs and global semantics

In this section, we describe how locations interact with each other, in order to allow the intuition presented above. To achieve that, we define a *global semantics*. This semantics is such that, when looking at a single location of the location graph, the behaviour of that location is consistent with the local semantics, even though it evolves within a whole location graph.

Informal presentation. The global semantics is not as intuitive as the local one. Before diving into its formal presentation, we present the underlying intuition.

The intended behaviour is the following: a location builds a list of possible transitions and, at some point, submits this list. After some time, when one of the possible transition is matched, the transition is selected and fired, and the other are forgotten.

In the meantime, we adopt a *lock mechanism*, which can temporarily freeze a location which attempts transitions until we are sure one of the transitions is allowed.

Aside, multiple (potentially independent) lists of transitions may be attempted by other locations, leading to multiple frozen locations simultaneously. In order to keep track of which list is related to which location, we use some identifiers which we associate to both the list and the (frozen) location which is

waiting for it.

Now, we have to identify when a list of pending transitions can be unfrozen. We define *complete* sets of transitions, which are, intuitively sets where all effects are matched (e.g. all $\mathit{expect}(\cdot,\cdot)$, $\mathit{send}(\cdot,\cdot)$, etc.). Then, we have to pick at most one transition of each list of possible transitions submitted, and see if it forms a complete set.

When we have selected a set of transitions to fire, we have to perform them (or at least, perform their effects on the skeleton location graph) before we can enqueue new pending transition: indeed, since the skeleton location may change, then the authorisation function may forbid previously allowed transitions, which then have to be cancelled. Therefore, when we discover a complete set of pending transitions, we split the set of pending transition into *selected* transitions (those in the complete set) and *unselected* transitions. For unselected transition, we (re)evaluate the authorisation function (with the updated skeleton location graph) and depending on the result, we abort or propose them again.

Preliminary definitions. Let begin by defining our notion of identified transitions. From the previous section, let remember that each location builds a list of transitions, which are then submitted to the system. When they are submitted, the system associate that list an identifier, which is just an integer. We call identified list of transition a tuple formed of that identifier and that list.

Definition 106 (Identified list of transitions)**.** An identified list of transitions is a tuple $\langle i, l\rangle$ where $i \in \mathbb{N}$ and $l \in \mathcal{P}^+(\mathbb{T})$.

Remark. In our case, two transitions never have the same identifier. Therefore, even if the two locations submit the same transition (e.g. both locations change their sort to the same sort), they will have different identifiers. This is the reason why, in the following, we only deal with *sets* of identified transitions and not with *multisets*. ◁

Similarly, an identified transition is an identifier associated with a (single) transition:

Definition 107 (Identified transition)**.** An identified transition is a tuple $\langle i, t\rangle \in \mathbb{N} \times \mathbb{T}$.

A second preliminary definition is the notion of complete multiset of transition. This notion will be used below to evaluate, after picking one transition of each proposed list, whether each $\texttt{Send}(\_,\_,\_)$, $\texttt{Expect}(\_,\_,\_)$, or $\texttt{Receive}(\_,\_)$ transition item of each transition we picked is matched by an other transition item.

Since $\oplus$ removes those matching transition items, checking if a multiset of transitions is complete is easy: we simply have to sum all items and verify that no such item remains.

Definition 108 (Complete set of transition)**.** Given a multiset of transitions $T \in \mathcal{P}^\star(\mathbb{T})$, the predicate complete$(T)$ is defined as:

$$\mathrm{complete}(T) \triangleq \neg\exists i \in \sum_{t\in T}^{\oplus} t \cdot i = \texttt{Expect}(\_,\_,\_) \vee i = \texttt{Receive}(\_,\_) \vee i = \texttt{Send}(\_,\_,\_)$$

We continue with a predicate used to pick some transitions out of a set of identified lists of transitions. The motivation is that we will have multiple locations, each of them submitting an identified list of transition, and we want to pick zero or one transition from each of these lists. Instead of writing a function that compute an actual selection, we define the following predicate, which, given a set of identified lists of transitions and a set of transitions, holds if and only if the given set of transitions is a correct selection of transitions from the identified lists.

Definition 109 (Picking relation)**.** Given a set of identified lists of transitions $l \in \mathcal{P}(\mathbb{N} \times \mathcal{P}^+(\mathbb{T}))$ and a set of identified transitions $s \in \mathcal{P}(\mathbb{N} \times \mathbb{T})$:

$$\mathrm{pick}(l, s) \triangleq \begin{cases} \text{true} & \text{if } s = \emptyset \\ \mathrm{pick}(tl, ts) & \text{if } (l = \{\langle i, \ell\rangle\} \cup tl) \wedge (s = \{\langle i, t\rangle\} \cup ts) \wedge (t \in \ell) \\ \text{false} & \text{otherwise} \end{cases}$$

This definition might look a bit convoluted but is in fact quite simple: if the set of selected transitions s is empty, then, of course, it is indeed a selection of zero or one transition from each list of proposed

transitions in l (actually, we pick zero transition from each list). Otherwise, we look for a transition $t \in s$ which is also in the corresponding set (i.e. the set with the same identifier i). If such t exists, then we have picked the transition for the i, and we continue for all other transitions in the remaining of l and s.

Remark. Assuming that identifiers are distincts, no two elements of l can refer to the same element of s and vice-versa. ◁

Remark. There is a possibility that no transition is picked from a given list: in that case, $l \neq \emptyset$ when we reach $s = \emptyset$. ◁

Example 11 (Picking transitions)**.** Consider that we have two proposed lists of transitions, each containing a single transition (note that those transitions are based on the synchronous counters of the previous examples):

$$l = \{\langle 0, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}]\rangle, \\ \langle 1, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}]\rangle\}$$

We want to verify that it is possible to select both transitions:

$$s = \{\langle 0, \{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}\rangle, \\ \langle 1, \{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}\rangle\}$$

In that case, the predicate holds:

$$\begin{aligned} \text{pick}(l, s) = \ & \text{pick}(\{\langle 1, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}]\rangle\}, \\ & \langle 1, \{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}\rangle) \\ = \ & \text{pick}(\emptyset, \emptyset) \end{aligned}$$

We define the function exchange(T) which takes a set of (selected) identified transitions (i.e. transition with their counter), and returns a set of tuples $\langle i, t, v\rangle$, where t is a transition, i its identifier, and v a multiset of tuples $\langle r, m\rangle$ with r a role identifier and m a message.

Definition 110 (Exchange function)**.** Given a set of identified transitions $T \in \mathcal{P}(\mathbb{N} \times \mathbb{T})$:

$$\text{exchange}(T) \triangleq \bigcup_{\langle i,t\rangle \in T} \langle i, t, \text{exchange}_t(\langle i, t\rangle, T)\rangle$$

with

$$\begin{aligned} &\text{exchange}_t(\langle i, t\rangle, T) \triangleq \\ &\quad \{\!\{\langle r, m\rangle | (\exists t_i \in t \cdot t_i = \texttt{Receive}(r, \_)) \wedge (\exists t' \in T \setminus \{\langle i, t\rangle\} \cdot \exists t'_i \in t' \cdot t_i = \texttt{Send}(r, m, \_))\}\!\} \end{aligned}$$

The predicate roles_ok$(\cdot, \cdot)$ ensures that roles bindings and releasings in the selected transition are allowed, that is it ensures the absence of double binding or releasing.

Definition 111 (Role predicate)**.** Given a set of identified transitions T and a skeleton location graph sk:

$$\begin{aligned} \text{roles\_ok}(T, sk) \triangleq & \{\!\{t_i | t_i \in \text{items}(T) \wedge t_i = \texttt{Bind}(\_, \_, \_)\}\!\} \text{ is a set} \\ & \wedge \forall \texttt{Bind}(\langle s, e\rangle, r, d) \in \text{items}(T) \cdot \neg\exists \langle r, d\rangle \in e \\ & \wedge \forall \texttt{Release}(\langle s, e\rangle, r, d) \in \text{items}(T) \cdot \exists \langle r, d\rangle \in e \\ & \wedge \forall \texttt{Bind}(s, r, d) \in \overset{\oplus}{\sum_{\langle i,t\rangle \in T}} t \cdot \neg\exists \langle s', e'\rangle \in sk \cdot \langle r, d\rangle \in e \end{aligned}$$

Where $\text{items}(T) \triangleq \{\!\{t_i | \exists \langle \_, t\rangle \in T \cdot t_i \in t\}\!\}$ is the multiset of transition items in transitions of the identified transition set T.

Remark. The first condition states that no two transitions can bind simultaneously the same role in the same direction. The second states that a location can not bind a role it has already bound. The third condition is the dual: a location can only only release roles it has bound. Finally, the last one states that each $\texttt{Bind}(s, r, d)$ must either bind a role which is simultaneously released; or a role that is not currently bound. ◁

Applying transitions on a skeleton location graph. Now that we have all the helper functions we need, we can define the effects of a transition on a skeleton location graph. First, we define the local effects on a single skeleton location (i.e. how the transition modifies the sort and the roles of the skeleton location), then we consider the global effects (i.e. we consider the creation and removal effects of that given transition). Finally, we consider the effects of a set of (identified) transitions on the skeleton location graph.

Let start with a function to evaluate the effects of a transition on a skeleton location. By reusing $\text{sort}(\cdot,\cdot)$ and $\text{endpoints}(\cdot,\cdot)$ defined above, such function is easily defined.

Definition 112 (Transition application to a skeleton location)**.** Given a transition $t \in \mathbb{T}$ and a skeleton location $sk \in \mathbb{\Sigma}_l$:

$$\text{apply}_t(\langle s, e\rangle, t) \triangleq \langle \text{sort}(s,t), \text{endpoints}(e,t)\rangle$$

To consider the creation and removal of skeleton location (i.e. to apply $\texttt{Remove}(\cdot)$ and $\texttt{Create}(\cdot)$ transition items of a transition), we define the function $\text{apply}(\cdot,\cdot)$ which takes a skeleton graph and a single transition and returns the modified skeleton graph and the remaining transitions items.

Definition 113 (Transition application to a skeleton location graph)**.** Given a transition $t \in \mathbb{T}$ and a skeleton location sk in $\mathbb{\Sigma}_l$, the function $\text{apply}(sk, t)$ returns a skeleton location graph in $\mathbb{\Sigma}$ according to:

$$\text{apply}(sk,t) = \begin{cases} \{\langle s, \emptyset\rangle\} \cup \text{apply}(sk, t') & \text{if } t = \{\texttt{Create}(s)\} \cup t' \\ \emptyset & \text{otherwise, if } \texttt{Remove}(sk) \in t \\ \text{apply}_t(sk,t) & \text{otherwise} \end{cases}$$

Remark. Notice that the evaluation order is important: if t contains both $\texttt{Create}(\cdot)$ and $\texttt{Remove}(\cdot)$ items, we want to perform the removal at the last (i.e. deepest) recursive call. ◁

This function allows to evaluate the effects of a transition on the related skeleton location. Our last step is to apply the effect of all selected transitions on the (global) skeleton location. We define the function $\text{apply}_f(\cdot,\cdot,\cdot)$ which applies the full set of (identified and selected) transitions to the global location graph. Notice that we need the set of frozen locations in order to lookup the right skeleton location for each transition.

Definition 114 (Full transition application to a skeleton location graph)**.** Given a skeleton graph $sk \in \mathbb{\Sigma}$, a set of identified transitions $T \in \mathcal{P}(\mathbb{N} \times \mathbb{T})$, and l a set of (potentially frozen) locations such that $\forall\langle i, t\rangle \in T \cdot \exists l^i \in l$ and such that $\forall\langle \sigma, \langle s, t, e, r\rangle\rangle^i \in l \cdot \exists\langle s, e\rangle \in sk$.

$$\text{apply}_f(sk, T, l) = \begin{cases} (\text{apply}_f(sk, T', l) \setminus \langle s, e\rangle) \cup \{\text{apply}(\langle s, e\rangle, t_i)\} & \text{if } T = \{\langle i, t_i\rangle\} \cup T' \\ & \quad \text{and } \langle \sigma, \langle s, t, e, r\rangle\rangle^i \in l \\ sk & \text{if } T = \emptyset \end{cases}$$

Remark. The two constraints on sk, l, and T are here to ensure that each transition in the selected set is associated with a frozen location, which itself has an adequate skeleton location in the skeleton location graph.

In our case, these conditions are always verified. ◁

Notice that for each t_i, we perform the recursive call to $\text{apply}_f(\cdot,\cdot,\cdot)$ prior to applying t_i. This is maybe a bit less intuitive than first applying t_i, then doing the recursive call on the modified skeleton graph, but it offers an advantage:after applying t_i to sk (to get say sk'), it may not be

the case anymore that $\forall \langle \sigma, \langle s, t, e, r\rangle\rangle^i \in l \cdot \langle s, e\rangle \in sk'$: indeed, the sort or the endpoints may have change; to solve that, we would need to remove l^i from l in the recursive call, which is doable but cumbersome.

Applying a transition on a location. Since locations can submit multiple transitions, we have to give a feedback for each transition. For each transition, the feedback can be that: (i) the transition is not in the set of unconstrained location transitions; (ii) the transition does not satisfy the authorisation function; (iii) the transition could be accepted, but was not (because an other proposed transition was selected instead); and (iv) the transition was selected, in which case the result is given.

We define the function $\text{result}(ts, ts, af, t, res, sk, l)$, which takes (among other things) a list of transitions ts, among which one (t) is selected; and returns a list of results.

Definition 115 (Results of a list of transitions)**.**

$$\text{result}(ls, ts(\cdot,\cdot), af(\cdot,\cdot,\cdot), t, res, sk, l) \triangleq$$
$$\begin{cases} \lceil \texttt{Not\_In\_Trans\_Set} \rceil :: \text{result}(tl, ts, af, t, res, sk, l) & \text{if } hd :: tl = ls \wedge \neg ts(hd, l) \\ \lceil \texttt{Not\_allowed} \rceil :: \text{result}(tl, ts, af, t, res, sk, l) & \text{otherwise, if } hd :: tl = ls \wedge \neg af(sk, l, hd) \\ res :: \text{result}(tl, ts, af, t, \lceil \texttt{Not\_Selected} \rceil, sk, l) & \text{otherwise, if } t :: tl = ls \\ \lceil \texttt{Not\_Selected} \rceil :: \text{result}(tl, ts, af, t, res, sk, l) & \text{otherwise, if } hd :: tl = ls \\ [] & \text{otherwise} \end{cases}$$

Remark. The seven parameters of the function are, in order: (i) the list of proposed transitions; (ii) the predicate that state whether a transition is in the set of unconstrained location transitions; (iii) the authorisation function predicate; (iv) the selected transition; (v) the result of the selected transition; (vi) the current skeleton graph of the system; and (vii) the location that takes the transition. Notice that, since a transition can appear twice in the list, and in particular the selected transition t, we only want to return res for the first one. Therefore, when the selected transition is found for the first time in the list, we then replace res by $\lceil \texttt{Not\_Selected} \rceil$, for eventual subsequent equal transitions.

Also, sk and l are present because they are required to evaluate the two predicates ts and af, but they are not modified. ◁

Remark. The order of the result matches the order of transitions. We rely on that for the user to find the result of each transition they proposed. ◁

One the other hand, we want to compute the effects of a transition on the (non-skeleton) location.

Definition 116 (Application of a transition to a location)**.** Given a location $l = \langle s, t, e, r\rangle$ and res is a set of $\langle r, m\rangle$ with r a role identifier and m a message.

$$\text{apply}_l(l, res, \sigma') = \begin{cases} \{\langle \sigma_i, \langle s, \emptyset, \emptyset, \lfloor \emptyset \rfloor\rangle\rangle\} \cup \text{apply}_l(\ell, \langle t', res\rangle, \sigma') & \text{if } t = \{\texttt{Create}(s)\} \cup t' \\ \emptyset & \text{otherwise, if } \texttt{Remove}(\_) \in t \\ \{\langle \sigma', \langle \text{sort}(s,t), \emptyset, \text{endpoints}(e,t), res\rangle\rangle\} & \text{otherwise} \end{cases}$$

This function takes a result res and a new state σ', and apply the global effects, of the transition of the location, sets its result, and updates its state.

Remark. Notice that, by convention, the initial value in the return field of a new location is $\lfloor \emptyset \rfloor$. ◁

Notice that the above function allows transitions that contains $\texttt{Remove}(\_)$ items mixed with other items. In practice, in the implementation, location removal will always be on a separate transition. This choice is arbitrary.

Location graph structure. The graph is a global state which is a tuple $\langle p, c, sk, l, i\rangle$ where p is a set of pending lists of transitions (associated with an identifier, formally, it is an element of $\mathcal{P}(\mathbb{N} \times \mathcal{P}^+(\mathbb{T}))$), c is a tuple of *selected* transitions and *unselected* transition lists (more formally, it is an element of $\mathcal{P}(\mathbb{N} \times \mathbb{T} \times \mathcal{P}^\star(\mathbb{R}, \mathbb{M})) \times \mathcal{P}(\mathbb{N} \times \mathcal{P}^+(\mathbb{T}))$), sk is a skeleton graph (an element of $\mathbb{\Sigma}$), l is a multiset of (potentially

frozen) locations and $i \in \mathbb{N}$ is a counter, used to avoid duplicated identifiers. We also assume we are given an authorisation function $af(sk, l, t)$ which is a predicate over skeleton location graphs, locations, and transitions and a predicate $ts(l, t)$ which states whether a transition t applied to the location l is in the set of unconstrained location transitions.

In the global view of the abstract machine, we decorate locations with an unique transition identifier to represent frozen locations associated with the used identifier (e.g. $\langle s, t, e, r\rangle^{id}$ for a frozen location that waits for the decision about the transition id).

A location graph is well formed if all its locations are well-formed, according to Definition 102, and if no two locations bind the same role in the same direction.

Definition 117 (Well-formed location graph)**.**

$$\begin{aligned}\mathtt{WF}_g(\langle p, c, sk, l, i\rangle) \triangleq\ & \forall\langle\sigma, h\rangle \in l \cdot \mathtt{WF}_h(h) \\ & \wedge \forall\langle\sigma_1, \langle s_1, t_1, e_1, r_1\rangle\rangle, \langle\sigma_2, \langle s_2, t_2, e_2, r_2\rangle\rangle \in l \cdot \forall e \in \mathbb{E} \cdot e \in e_1 \Rightarrow e \notin e_2\end{aligned}$$

Semantics. The first rule of our global semantics simply maps the local changes of locations. Said otherwise, it simply lift the local rules (Send), (Expect), (Sort), (Sort), (Bind), (Release), (Create), (End), (Silent), and (Reset) to the global semantics.

$$\text{(GSilent)}\ \frac{\ell \to \ell'}{\langle p, c, sk, \{\ell\} \cup l, i\rangle \to \langle p, c, sk, \{\ell'\} \cup l, i\rangle}$$

Now, we implement the mechanism that performs location transitions. As stated in the informal presentation, the first step is to add the transition to the set of pending transitions and to freeze the appropriate location.

$$\text{(Add)}\ \frac{\ell \xrightarrow{t} \ell'}{\langle p, \langle\emptyset, \emptyset\rangle, sk, \{\ell\} \cup l, i\rangle \to \langle p \cup \{\langle i, t\rangle\}, \langle\emptyset, \emptyset\rangle, sk, \{\ell^i\} \cup l, i + 1\rangle}$$

In (Add), notice that the location ℓ does not take its transition. Informally, the rule says: "If a location ℓ can take a transition t, then t is proposed and ℓ locks". Notice also that it is possible to fire (Add) only if the chosen transitions is $\langle\emptyset, \emptyset\rangle$[17].

Once it is possible to pick a complete set of transition from the set of lists of transitions, we can select it. During this selection, message are also exchanged (function exchange).

$$\text{(Select)}\ \frac{\text{pick}(p, s) \qquad T = \{t | \langle i, t\rangle \in s\} \qquad \text{complete}(T) \qquad \text{roles\_ok}(T, sk)}{\langle p, \langle\emptyset, \emptyset\rangle, sk, l, i\rangle \to \langle\emptyset, \langle\text{exchange}(s), p \backslash s\rangle, \text{apply}_f(sk, p, l), l, i\rangle}$$

Finally, once transitions are selected, each location can perform it, and mirror the effects on the skeleton graph:

$$\text{(Perform)}\ \frac{\text{next}(\langle\sigma, \langle s, t, e, r\rangle\rangle) = \langle\_, \sigma'\rangle}{\begin{array}{l}\langle p, \langle s \cup \{\langle i, t, m\rangle\}, r\rangle, sk, l \cup \{\langle\sigma, \langle s, t, e, r\rangle\rangle^i\}, j\rangle \to \\ \langle p, \langle s, r\rangle, sk, l \cup \text{apply}_l(\langle\sigma, \langle s, t, e, r\rangle\rangle, \text{result}(t, ts(\cdot, \cdot), af(\cdot, \cdot, \cdot), t, \lfloor m\rfloor, sk, \langle\sigma, \langle s, t, e, r\rangle\rangle), \sigma'), j\rangle\end{array}}$$

For transitions that were not selected, the transition is retried. Notice that rule (Select) requires the set of selected and unselected transitions to be $\langle\emptyset, \emptyset\rangle$. This prevents it to fire to early.

$$\text{(Retry)}\ \frac{}{\langle p, \langle s, r \cup \{\langle i, t\rangle\}\rangle, sk, l \cup \{\ell^i\}, j\rangle \to \langle p \cup \{\langle j, t\rangle\}, \langle s, r\rangle, sk, l \cup \{\ell^j\}, j + 1\rangle}$$

Example 12 (Synchronous integer counters (continued))**.** We consider the location graph with two locations such as presented in Example 10, with $i = 0$ and $i = 1$. Initially, the location graph is the following (with the transition counter starting arbitrarily at 0, and the initial n being arbitrarily

[17] If there are some transitions in the chosen set, it means the previous transition is not resolved yet.

chosen to be $n = 1$):

$$\langle \emptyset, \langle \emptyset, \emptyset \rangle, \{\!\{ \langle \top, \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}\rangle, \langle \top, \{\langle r_1, \texttt{provided}\rangle, \langle r_0, \texttt{required}\rangle\}\rangle \}\!\},$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{i})\rangle, \langle \top, [], \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}, \_\rangle\rangle,$$
$$\langle\langle 1, 1, (\mathrm{i})\rangle, \langle \top, [], \{\langle r_1, \texttt{provided}\rangle, \langle r_0, \texttt{required}\rangle\}, \_\rangle\rangle \}\!\}, 0\rangle$$

For the sake of readability, we write sk for

$$\{\!\{ \langle \top, \{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}\rangle, \langle \top, \{\langle r_1, \texttt{provided}\rangle, \langle r_0, \texttt{required}\rangle\}\rangle \}\!\}$$

e_0 for

$$\{\langle r_0, \texttt{provided}\rangle, \langle r_1, \texttt{required}\rangle\}$$

and e_1 for

$$\{\langle r_1, \texttt{provided}\rangle, \langle r_0, \texttt{required}\rangle\}$$

These two locations can build up their transitions. From a global perspective, it is a sequence of (GSilent) rules. Notice that the order we choose in this example is a possibility among others.
First, the location with $i = 0$ can take a (Send) transition, which is allowed with the (GSilent) rule of the global semantics. Its local state changes:

$$\langle \emptyset, \langle \emptyset, \emptyset \rangle, sk,$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{ii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}) \}\!\}], e_0, \_\rangle\rangle,$$
$$\langle\langle 1, 1, (\mathrm{i})\rangle, \langle \top, [], e_1, \_\rangle\rangle \}\!\},$$
$$0\rangle$$

The second location can do the analog change:

$$\langle \emptyset, \langle \emptyset, \emptyset \rangle, sk,$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{ii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}) \}\!\}], e_0, \_\rangle\rangle,$$
$$\langle\langle 1, 1, (\mathrm{ii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_1, 1, \texttt{provided}) \}\!\}], e_1, \_\rangle\rangle \}\!\},$$
$$0\rangle$$

The local semantics allow the first location to do a (Receive) transition. This can be lifted to the global semantics, resulting in the following graph:

$$\langle \emptyset, \langle \emptyset, \emptyset \rangle, sk,$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{iii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required}) \}\!\}], e_0, \_\rangle\rangle,$$
$$\langle\langle 1, 1, (\mathrm{ii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_1, 1, \texttt{provided}) \}\!\}], e_1, \_\rangle\rangle \}\!\},$$
$$0\rangle$$

And finally, the second location can perform the analog change:

$$\langle \emptyset, \langle \emptyset, \emptyset \rangle, sk,$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{iii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required}) \}\!\}], e_0, \_\rangle\rangle,$$
$$\langle\langle 1, 1, (\mathrm{iii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required}) \}\!\}], e_1, \_\rangle\rangle \}\!\},$$
$$0\rangle$$

Then, two global (Add) rules can be taken (we assume the authorisation function holds). The order is arbitrary. In particular, the (Add) transition of one of the two locations can be interleaved with the (GSilent) transition (above) of the other.
The first transition gives the following graph:

$$\langle \{\langle 0, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required}) \}\!\}]\rangle\}, \langle \emptyset, \emptyset \rangle, sk,$$
$$\{\!\{ \langle\langle 0, 1, (\mathrm{iii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required}) \}\!\}], e_0, \_\rangle\rangle^0,$$
$$\langle\langle 1, 1, (\mathrm{iii})\rangle, \langle \top, [\{\!\{ \texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required}) \}\!\}], e_1, \_\rangle\rangle \}\!\},$$
$$1\rangle$$

And after the second transition:

$$\begin{aligned}
&\langle \{\langle 0, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}]\rangle,\\
&\ \ \langle 1, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}]\rangle\},\\
&\langle \emptyset, \emptyset\rangle, sk,\\
&\{\!\{\langle\langle 0, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}], e_0, _\rangle\rangle^0,\\
&\ \ \langle\langle 1, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}], e_1, _\rangle\rangle^1\}\!\},\\
&2\rangle
\end{aligned}$$

At this point, note that $\sum^{\oplus}_{t \in T} t = \emptyset$, for

$$\begin{aligned}
T = \{&\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\},\\
&\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}\}
\end{aligned}$$

Therefore complete(T) holds. Also, there is no $\texttt{Bind}(\_, \_, \_)$ nor $\texttt{Release}(\_, \_, \_)$ in T, therefore roles_ok(T, sk) trivially holds. Finally, from Example 11, pick(l, s) holds.
Therefore, a (Select) transition can be taken. Notice that, if the skeleton graph had to be changed, it would be done atomically in this transition.

$$\begin{aligned}
&\langle \emptyset, \langle \{\langle 0, \{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}, \{\!\{\langle r_1, 1\rangle\}\!\}\rangle,\\
&\qquad \langle 1, \{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}, \{\!\{\langle r_0, 1\rangle\}\!\}\rangle\},\\
&\ \ \ \emptyset\rangle,\\
&sk,\\
&\{\!\{\langle\langle 0, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_0, 1, \texttt{provided}), \texttt{Receive}(r_1, \texttt{required})\}\!\}], e_0, _\rangle\rangle^0,\\
&\ \ \langle\langle 1, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}], e_1, _\rangle\rangle^1\}\!\},\\
&2\rangle
\end{aligned}$$

Now, each location can perform its (Perform) transition. Again, the order is arbitrary. For instance, if the location with $i = 0$ does it first, it leads to the following graph:

$$\begin{aligned}
&\langle \emptyset,\\
&\langle \{\langle 1, \{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}, \{\!\{\langle r_0, 1\rangle\}\!\}\rangle\}, \emptyset\rangle,\\
&sk,\\
&\{\!\{\langle\langle 0, 1, (\text{iv})\rangle, \langle \top, [], e_0, \lfloor \{\!\{\langle r_1, 1\rangle\}\!\} \rfloor\rangle\rangle,\\
&\ \ \langle\langle 1, 1, (\text{iii})\rangle, \langle \top, [\{\!\{\texttt{Send}(r_1, 1, \texttt{provided}), \texttt{Receive}(r_0, \texttt{required})\}\!\}], e_1, _\rangle\rangle^1\}\!\},\\
&2\rangle
\end{aligned}$$

And after the second (Perform):

$$\langle \emptyset, \langle \emptyset, \emptyset\rangle, sk, \{\!\{\langle\langle 0, 1, (\text{iv})\rangle, \langle \top, [], e_0, \lfloor \{\!\{\langle r_1, 1\rangle\}\!\} \rfloor\rangle\rangle, \langle\langle 1, 1, (\text{iv})\rangle, \langle \top, [], e_1, \lfloor \{\!\{\langle r_0, 1\rangle\}\!\} \rfloor\rangle\rangle\}\!\}, 2\rangle$$

Finally, we arrive at a point where locations can take over with their local transitions. Thus, by firing (GSilent) twice, we arrive at the initial point, with n increased on both locations.
The first (GSilent) leads to:

$$\langle \emptyset, \langle \emptyset, \emptyset\rangle, sk, \{\!\{\langle\langle 0, 2, (\text{i})\rangle, \langle \top, [], e_0, \lfloor \{\!\{\langle r_1, 1\rangle\}\!\} \rfloor\rangle\rangle, \langle\langle 1, 1, (\text{iii})\rangle, \langle \top, [], e_1, \lfloor \{\!\{\langle r_0, 1\rangle\}\!\} \rfloor\rangle\rangle\}\!\}, 2\rangle$$

And the second to:

$$\langle \emptyset, \langle \emptyset, \emptyset\rangle, sk, \{\!\{\langle\langle 0, 2, (\text{i})\rangle, \langle \top, [], e_0, \lfloor \{\!\{\langle r_1, 1\rangle\}\!\} \rfloor\rangle\rangle, \langle\langle 1, 2, (\text{i})\rangle, \langle \top, [], e_1, \lfloor \{\!\{\langle r_0, 1\rangle\}\!\} \rfloor\rangle\rangle\}\!\}, 2\rangle$$

At this point, we have to establish a correspondance between the calculus presented in Chapter 3 and the abstract machine we have presented in this section.

Two main difficulties have to be resolved in order to achieve this correspondance: (i) managing the fact that transitions in the abstract machine are not atomic; and (ii) studying the newly introduced errors of the abstract machine.
The first difficulty is that, in the abstract machine, transitions are not taken at a single point: each location has to perform an individual (PERFORM) transition, and possibly some (GSILENT) transitions. Therefore, there are some points in the execution of the abstract machine, where the states of individual locations are not consistent with each other.
Fortunatelly, we know that, to select a set of transitions, all transitions selected previously must be performed. Therefore, the starting state of a (SELECT) transition must be consistent. Therefore, we must be able to find a correspondance at these points.
Therefore, I think the correspondance can be done in two steps: (i) we show that, for each (initially correct) execution of the abstract machine, at each starting state of a (SELECT) transition, the stored skeleton graph is consistent with the locations; and (ii) we show that there is a correspondance between the skeleton graph of the theoretical framework and the skeleton graph stored in the abstract machine.
The second difficulty concerns the errors that are introduced (Definition 96): the theoretical framework does not have this notion of error, they either take a transition or hang if no transition is possible[a]; while, in the abstract machine, a location can attempt a transition, which fails.
The question is: how do that behaviour of the abstract machine relate to the theoretical framework? Fortunatelly, a failed transition has no impact on the stored skeleton graph, and the failure is decided at the time of the selection of the transitions to fire. Therefore, if a location fails a transition in the abstract machine, it can simply be a (local) transition of the theoretical framework, where only the process state changes. Therefore, in the theoretical framework, we could probably extend the set of transition to simulate failed attempts.
All in all, one could expect to show that the abstract machine *simulates* the theoretical framework, and that for all instance of the abstract machine, it is possible to find a theoretical instance which simulates that instance of the abstract machine.
The work to formalise the two intuitions to accomodate those two differences is left to do. We suppose that the intuition we developed are correct.

[a]Said otherwise, locations in the theoretical framework follow Yoda's advice: "Do, or do not. There is no try". (*The empire stricks back*, George Lucas, 1980)

5.3 Rust API

In this section, we explain informally the API of our library, from the programmer's perspective.

The location graph framework is such that location behaviour can be written independently from the authorisation function specification. Therefore, this section is organised as follows: Subsection 5.3.2 and 5.3.3 respectively describe how to program locations and authorisation functions and are independent.

In order to illustrate this section, we follow up on the synchronous counter example presented in the previous section.

For the user, the location framework comes as a Rust library[18]: *rust_locations*. It is intended to be used as a replacement for standard actor systems, such as the Scala library Akka [1, 59]. In Akka, the user instanciates an *ActorSystem* which manages the different *Actors*. Similarly, in our implementation, the user instanciates an *LocationSystem*, which manages the *Locations*. To take into account the specifities of locations over actors, as well as differences between Scala and Rust, our library has a few differences:

- In Location Graphs, transitions are synchronous, while Actor communications are not.
- Locations have *sorts*, therefore, most structures have a generic type `S` which is the type of Sorts. Similarly, the library is generic over messages (type `M`) and role identifiers (type `R`).
- Transitions are allowed or forbidden according to an *authorisation function*. The user must provide a structure that implements the `AuthorisationFunction` trait[19].
- The instanciation of a *LocationSystem* requires an instance of a structure that implements the `AuthorisationFunction` trait.
- Rust favors *composition over inheritance*[20]. Therefore, while, in Akka, the user implements classes

[18]A Rust crate, using Rust terminology.
[19]The Rust language uses *traits* which are more or less equivalent to interfaces in other languages.
[20]There is no notion of structure/class inheritance in Rust. In practice, we tend to compose structures (i.e. include a structure as a field of an other structure). A notion of inheritance exists for traits.

`send(&mut self, r: R, msg: M)`	Sends the message `msg` on role `r` during the next transition.
`receive(&mut self, r: R)`	Receives a message on role `r` during the next transition.
`expect(&mut self, r: R, msg: M)`	During the next transition, performs a synchronisation with message `msg` on role `r`.
`bind(&mut self, r: R, d: RoleDirection)`	Binds the role `r` in direction `d` during the next transition.
`release(&mut self, r: R)`	Releases the role `r` during the next transition.
`change_sort(&mut self, s: S)`	Changes the sort to `s` during the next transition.
`create(&mut self, s: S)`	During the next transition, create a new `Location` with `s` as initial sort.
`commit_changes(&mut self)`	Attempt to perform the transition. This primitive returns the result of the transition or an error.

Figure 5.2: Primitives of locations provided by the `Location` structure.

that inherits the `Actor` class, in our Rust library, the user manipulates locations which provide primitives to take transitions.

5.3.1 Preliminary steps

We first have to choose the basic types of our instance. The library impose very few constraints on the basic types used for sorts, messages, and role names.

In our example, sorts are not used, so we can use the `()` type. Messages are integers, we will use the Rust primitive type `i32`. Finally, we need some type for role names, since we need to create some roles. However, role names do not serve any purpose. Therefore, we use the opaque type `DefaultRole` provided by the library. This type is opaque, but we are provided a generator to instantiate as many such role names as required on the fly.

5.3.2 Locations

On one side of the library stands the `Location` structure. This structure is not well named: it does not represent a location of a location graph instance; it is rather a handle which provides the primitives to work with the library (it is the equivalent of location handles of the abstract machine, see Definition 101).

Since each instance of `Location` is a handle to access location primitives, each part[21] of a program that has a reference to such handle can be viewed as a location. Using the handle, locations can build a transition by accumulating items, and then fire this transition. The table in Figure 5.2 shows the primitives provided by the `Location` structure.

The only primitive that returns a value is `commit_changes`. The other are used to incrementally build a transition which is then fired by `commit_changes`. Each of the primitives presented correspond to the one of the abstract machine (see Definition 100). Firing a transition can fail for various reasons (e.g. if the transition is not allowed). If it succeed, then some values can be returned (e.g. some value received during a communication, or some instances of `Location` if a new locations are created).

Writing a single location is straightforward, it consists in a sequence of calls to primitives to build a transition, then firing and analysing the result.

Example 13 (Synchronous integer counters (continued))**.** The example in Figure 5.3 shows a simple location. This example illustrates the implementation of one of location of our running example: the structure `PingPongLocation` contains the state of the location, as well as the location handle. Notice that the state does not exactly correspond to its abstract machine counterpart: the automaton state is not explicit, since the automaton is not explicitly implemented using a state machine, but with the program flow of the location. Also, the identifier is not explicitly given: it is only used to break location symmetry to disambiguate the role names. Here, locations are directly

[21] We are intentionally vague about what is a *part* of a program. It can be a thread, a structure, etc. .

given the role names they use, and therefore they do not need to store an identifier. However, we introduce a `name` field, for printing purposes. Finally, the structure also contains a `Location` field, to access the location primitives.
Consider the method `run` of the figure. This method implements the main loop of the location, i.e. it is the implementation of the automaton described in the previous parts of this example. In that method, the structure instanciates a new message (`let msg = self.i + 1`), and sends it to the other location via the roles with identifier `id1` (`self.l.send(id1, msg)`). In the meantime, it also adds the receive of the value from the other side on the role with identifier `id2` (`self.l.receive(id2)`). The transition is then ready, it is fired (`self.l.commit_changes()`).
The method `commit_changes` returns a list of results, one for each of the proposed transition. In this case, since the location proposed only one transition, only one result is expected. Therefore the first element of the list of result is taken (`pop().expect(...)`) and matched against the possible returned value (the `match` block). The value is either a success (`Ok(v)`) or a failure (`Err(e)`).
In case of a success, `v` contains the set of results of the selected transition (in our case, we are only expecting a single received value but, in general, this set contains one element per value received, and one per location created). For each element of this set (i.e. for the only element of this set), if that element is a message reception (the `if let TransitionResult::RECEIVED(_, msg) = res` line), then we consider the integer received and the local counter `self.i` is updated accordingly.
In case of failure, the error `e` is simply printed and the location ends.

5.3.3 Authorisation functions and unconstrained location transitions

The other side of the library is used to specify our set of unconstrained location transitions and to write our authorisation function. Notice that, except for the basic types that must be consistent with the one chosen above, this is independent from writing the location code.

Specifying the set of unconstrained location transitions is done using a predicate over locations and transitions which states whether the transition is in the set. We have a trait `TransitionSet` which requires a single method `contains` which implements this predicate.

Example 14 (Synchronous integer counters (continued))**.** In our example, the set of unconstrained location transitions can be as relaxed as possible; we therefore use the `TrivialTS` structure which accepts all transitions.
This set of unconstrained location transitions is provided by the library, and implements the biggest set of unconstrained location transitions (according to the chosen base types), i.e. the set that contains all unconstrained locations transitions. More details on this provided set is given in Section 5.5.1.

Writing an authorisation function is actually quite close to the theory: in the theoretical model, an authorisation function is a predicate on the skeleton graph and the transition to be taken. In the library, writing an authorisation function is simply writing a structure which implements the `AuthorisationFunction` trait. This traits requires a single function, which takes a skeleton graph, a location and a transition and returns a boolean. Notice that, in this context, a *transition* is a set of atomic modifications that the location built up before committing the transition.

Example 15 (Synchronous integer counters (continued))**.** In our case, to illustrate the concept, we want to allow a few transitions (say $n = 12$), and then forbid all transitions, except location removal.
In Figure 5.4, we implement the authorisation function that corresponds to such a policy. The authorisation function is stateful, to be able to count the transitions, and contains a single synchronised shared counter (the field `i: Arc<AtomicU64>`). This counter needs to be synchronised since there is an instance of the authorisation function structure per location, and each location is on a separate thread.
The authorisation function in itself is very simple: the method `authorise` checks whether the transition is the removal of the location (the function `authorise_removal`, provided by the library). If it is the case, the transition is accepted. Otherwise, the counter is incremented and the authorisation function returns true if its value is less 12 (the line `self.i.fetch_add(1, ...) <= 12`).
Note that the evaluation is lazy, and therefore, if `authorise_removal` returns true, the counter is not modified.

```rust
struct PingPongLocation {
    l: Location<DummySort, PingPongAF, i32, DefaultRole, TS>,
    i: i32,
    name: &'static str,
}

impl PingPongLocation {
    fn new(l: Location<DummySort, PingPongAF, i32, DefaultRole, TS>, name: &'static str) -> Self {
        PingPongLocation { l, i: 0, name }
    }

    fn run(&mut self, id1: DefaultRole, id2: DefaultRole) {
        self.initialise(id1, id2);
        loop {
            let msg = self.i + 1;
            self.l.receive(id2);
            self.l.send(id1, msg);
            match self.l.commit_changes()
                .pop()
                .expect("Could not find the result of the transition.")
            {
                Ok(v) => {
                    println!("[{}] Sent {}, received {}", self.name, self.i + 1, self.i);
                    for res in v {
                        if let TransitionResult::RECEIVED(_, msg) = res {
                            if self.i < *msg {
                                self.i = *msg
                            }
                        }
                    }
                }
                Err(e) => {
                    println!("[{}] Could not commit changes: {:?}", self.name, e);
                    return;
                }
            }
        }
    }

    fn initialise(&mut self, id1: DefaultRole, id2: DefaultRole) {
        self.l.bind(id1, RoleDirection::PROVIDED);
        self.l.bind(id2, RoleDirection::REQUIRED);
        if let Err(_) = self.l.commit_changes()
            .pop()
            .expect("Could not find the result of the transition.")
        {
            panic!("Couldn't bind");
        }
    }
}
```

Figure 5.3: Example of a simple location. This location loops indefinitely and, at each iteration, it sends a value a stored value (incremented by 1), receives an other one, and stores the new one if it is greater than the currently stored one. Notice that the structure `PingPongLocation` holds a structure `l` with type `Location` which provides access to the primitives, e.g. `receive`, `send`, `bind` and `commit_changes`. This `Location` structure is generic over some types, namely the sort used (here `DummySort`, which is just an alias for `()`), the authorisation function (here `PingPongAF`, explained below), the type of the messages (here `i32`) and the types of the role names (here `DefaultRole`, which is just a generic role identifier provided by the library).

```rust
struct PingPongAF {
    i: Arc<AtomicU64>,
}

impl AuthorisationFunction for PingPongAF {
    type Sort = DummySort;
    type Msg = i32;
    type RoleID = DefaultRole;
    type TransitionSet = TS;

    fn authorise(
        &mut self,
        _s: &SkeletonGraph<DummySort, DefaultRole>,
        t: &Transition<Self::Sort, Self::Msg, Self::RoleID>,
        _: &Location<DummySort, PingPongAF, i32, DefaultRole, TS>,
    ) -> bool {
        authorise_removal(t) || self.i.fetch_add(1, Ordering::Relaxed) <= 12
    }
}
```

Figure 5.4: Example of a simple authorisation function. This authorisation function simply has a counter `i` which is incremented each time the authorisation function is polled. When this counter is more than 12, transitions are forbidden. Notice that we exclude removal transition from this count. Notice also that the counter is a shared variable, indeed, there is one instance of the authorisation function per location, therefore shared variables should be explicit. Notice that, in theory, authorisation functions do not have a state, be it shared or not. In this implementation, it actually comes for free.

Browsing the skeleton graph. In the presented example, we did not need to explore the skeleton graph. In order to provide a more complete introduction to programming with location graphs, we will conclude this subsection by a word on those.

The `SkeletonGraph` structure is simply a set of `SkeletonLoc`. It provides methods to easily filter locations. In addition, we provide helper functions to walk on the graph, which is particularly useful in the case of the policies shown in previous chapters, where one has to express properties on paths.

To even ease the expression of path related conditions, we also implemented graph formulas, which allow to express simple formulas of modal logic. Graph formulas are built up from location formulas and role formulas. The former is basically a predicate on roles and the sort of a location, and the latter is a predicate on role names. Graph formulas are then formed using those predicate to express properties such as: the location has a role named a which reaches a location with sort s and a bound role r[22].

5.3.4 Final steps

On one side, we have our locations, and on the other side, we have our authorisation function. We now want to put it all together and run the instance.

In order to run the instance, the user has to instantiate manually the initial locations of the instance, and run them on separate threads. They also have to create an initial instance of the authorisation function.

Notice that the user shall not (and can not) instantiate locations by themself, since locations should also be registered by the system to construct the initial skeleton location. To create the first locations, the user creates an array[23] of sorts and gives this array at initialisation. In exchange, they receive an array of `Location`s structures, which can then be used.

Notice that locations have to be run on separate threads. We do not specify further the concurrency model (be it lightweight threads, async, etc.) we are talking about here, since we only make the assumption of an underlying concurrency model. In practice, this question should be addressed for performance issues.

Remark. All Rust examples presented are run using the standard lightweight threads of the `std::thread` library. ◁

[22] Graph formulas are not used in the following of this thesis, we only provide them as utilities for users. Therefore, we do not present them in more detail here.

[23] Actually, the user only needs to give an `Iterator` over sorts.

```rust
fn pingpong_test() {
    let af = PingPongAF::new();

    let initial_sorts = vec![(), ()];
    let mut locs = LocationSystem::new(af, TS::new(), initial_sorts.into_iter());

    let mut idg = DefaultRoleGenerator::new();
    let id1 = idg.new_id();
    let id2 = idg.new_id();

    let mut p1 = PingPongLocation::new(locs.pop().unwrap(), "ID1");
    let mut p2 = PingPongLocation::new(locs.pop().unwrap(), "ID2");

    let j1 = thread::spawn(move || p1.run(id1, id2));
    let j2 = thread::spawn(move || p2.run(id2, id1));

    j1.join().expect("j1␣crashed");
    j2.join().expect("j2␣crashed");
}
```

Figure 5.5: Example of the initialisation and run of our example. We first instantiate an authorisation function. We then create an array of which contains the sorts of the initial locations, and instantiate the overall `LocationSystem`, which returns the initial location handlers. These location handlers are used to instantiate our `PingPongLocation`s, which are finally run on separate threads, and we wait until the threads terminate.

Example 16 (Synchronous integer counters (last))**.** The initialisation of our running example is shown in Figure 5.5. First, we have to instantiate a first instance of our authorisation function, and an array which contains the sorts of the initial locations (the line `let initial_sorts = vec![() , ()]`, in our case, we use `()` for our sorts). We can then instantiate the location system, using the above objects as parameters, and an instance of the trivial unconstrained location transition set. Instantiating the location system returns a list containing the initial locations handles (one per initial sort provided, in the same order).
With the location handles, we can instantiate the locations, and run them on two threads.
The output of a typical run is given in Figure 5.6. Since we allow only the 12 first transitions, only 10 integers are exchanged, plus two initial transitions for binding roles. The subsequent transition are rejected, with the `NotAuthorised` error.

```
[ID2] Sent 1, received 0
[ID1] Sent 1, received 0
[ID1] Sent 2, received 1
[ID2] Sent 2, received 1
[ID2] Sent 3, received 2
[ID1] Sent 3, received 2
[ID1] Sent 4, received 3
[ID2] Sent 4, received 3
[ID2] Sent 5, received 4
[ID1] Sent 5, received 4
[ID1] Could not commit changes: NotAuthorised
[ID2] Could not commit changes: NotAuthorised
```

Figure 5.6: Typical output of our running example. Notice that there are 10 lines of output (corresponding to 10 transitions) before the unauthorised transitions, instead of the $n = 12$. This is due to the two (one for each location) silent initial transitions used to bind the roles.

5.4 Implementation

Section 5.3 should already give a good hint of the behaviour of our crate. In this section, we explore in depth the internal details of the implementation of the library.

To better understand this section, let us analyse a typical execution of a program using the location library.

We have a few threads, each running the function of a location (i.e. its process). To interact with other locations, the different functions must have access to the primitives highlighted in the previous section. Therefore, the first components of the library we will explain are `Location`s, which provide various methods to build and commit a `Transition`, the second element of the library.

When a transition is committed, the `Location` takes over the running function and cooperates with other locations to resolve the transition. To resolve the transition, an other structure is used: the `Sevaluator`[24]. Notice that each `Location` has a local instance of a `Sevaluator`. This structure evaluates if the transition is allowed and interacts with a shared memory area used to centralise all pending transitions. The `Sevaluator` selects a set of resolvable transitions, or waits until such a set is selected by an other `Sevaluator`. When a set of transitions is selected, the `SkeletonGraph` is updated accordingly; and each waiting `Sevaluator` checks whether its transition is selected, and, depending on the result, retries or resolve the transition.

Resolving the transition consists not only in creating new locations, but mostly in binding and releasing roles, as well as exchanging messages. These last two things are performed by the `Role`s and the `RoleManager` structures.

To evaluate if a transition is allowed, an `AuthorisationFunction` is provided to the system, which uses the current `SkeletonGraph`.

To wrap up this introduction, and to have a clear understanding of the library, let examine it from a memory viewpoint. The shared memory contains five objects: (i) the set of pending transitions; (ii) the set of selected transitions; (iii) the skeleton graph; (iv) the transition counter; and (v) the role manager. Each of these memory areas is protected by a mutex[25]. These objects are the only one that are shared[26].

Outline. As a first step, we will look at `Location`s and `Transition`s (Section 5.4.1). Our second step will be to study the implementation of `SkeletonGraph`s (Section 5.4.2). Then, we will explain the selection of `Transition`s performed by the `Sevaluator`s (Section 5.4.4), followed by the resolution of the selected `Transition`s; which requires a digression to `Role`s and the `RoleManager` beforehand (Section 5.4.3). Finally, we will explain is the `AuthorisationFunction` and `TransitionSet` traits in Section 5.4.5.

Remark (Non-trivial aspects of the implementation). Most of the implementation is a trivial transposition of what have been explained about the abstract machine into Rust. The main non-trivial point is Section 5.4.4 about transition selection and resolution. ◁

Through this section, we will highlight our discourse with excerpts from the library code. However, we will not cover every single line of code.

Remark. In the remaining of this section, the interested reader is encouraged to build the documentation with private items. In the root directory, run:

```
> cargo doc --document-private-items
```

◁

5.4.1 Locations and Transitions

Locations. Locations are the main aspect of location graphs. As in the abstract machine, in the implementation, we distinguish locations and location handles: a location is a structure to be implemented by the user. It works with an handle, which provides the primitives of the location framework.

Therefore, our library only offers location handles, via a structure (improperly) named `Location`. The structure is shown in Figure 5.7. This structure is a black box from the user standpoint[27].

Locations primitives. Locations have several methods. Most of them are self explanatory. The only non trivial detail, for programmers not used to Rust, is how exchange primitives interact with the Rust typesystem.

[24] Fundamentally, it performs the `seval()` function, hence its name.
[25] See the Appendix A on shared memory in Rust.
[26] The Rust typesystem ensures that it is not possible to *accidentally* share some memory: each object that is aliased must be explicitly aliased: using a `Arc` (if aliased among concurrent threads) or an `Rc` (if aliased in a single thread).
[27] Without documenting private items, the documentation contains only the primitives.

```rust
pub struct Location<S, A, M, R, T>
where
    S: Clone + Eq,
    A: AuthorisationFunction<Sort = S, Msg = M, RoleID = R, TransitionSet = T>,
    M: Send + Eq + Clone,
    R: RoleID,
    T: TransitionSet<Sort = S, AF = A, Msg = M, RoleID = R>,
{
    ls: LocationSystem<S, A, M, R, T>,
    roles: Vec<RoleEndpoint<M, R>>,
    sort: S,
    t: Vec<Transition<S, M, R>>,
}
```

Figure 5.7: The structure `Location` of the Rust library. Notice that all the fields are private (i.e. they are not annotated with `pub`), therefore they are not visible outside the library (e.g. by the user).

```rust
/// A trait for role identifier.
pub trait RoleID: Clone + Ord + Hash {}
```

Figure 5.8: The definition of the trait `RoleID`.

Take the example of `send` (`expect` is similar). Its signature is `pub fn send(&mut self, r: R, msg: M)`, with `R` the type of role identifiers and `M` the type of messages. This signature means that we borrow (take a mutable reference) to `self` (the `Location`), and that we *acquire* (we claim ownership of the memory area) both the role identifier and the message. That is, when calling the method `send`, the caller can not use the message after the call. This mechanism prevents aliases[28], and therefore ensures the message is indeed sent to the new location.

Notice that it does not prevent the message to be copiable[29], but even in this case, the message must be duplicated, which ensure that there is no aliases: the receiver would receive an other instance of the message.

Concerning the role identifier `r`, the definition of the trait `RoleID` requires that identifiers are copyable (see Figure 5.8). Therefore, the user can use multiple time the same role identifier.

Such mechanism, implemented *in* the Rust typesystem, allows us not to implement by ourself, such as what Sabah did in its thesis (see [50]).

Transitions and Transition items. `Location`s are structures that provide the user the primitives of the location graph framework. Via methods such as `send`, `bind`, etc., it is possible to build a transition, which is then taken using the method `commit_changes`.

Transitions (type `Transition`) are sets of transition items (type `TransitionItemType`); itself an enumerated type with eight variants, corresponding to the transition items of the abstract machine.

A transition describes all the changes that happen during the transition.

- `BIND(R, RoleDirection)`, to bind a new role in a given direction.
- `CREATE(Box<S>)`, to create a new location with a given initial sort. Notice that the location is created without bound roles. It is intended to be given to a new thread.
- `EXPECT(R, Box<M>)`, try to synchronise on the given role with the given value.
- `RECEIVE(R, RoleDirection)`, to wait to receive a value on the given role.
- `RELEASE(R)`, to release the given role.
- `SEND(R, Box<M>, RoleDirection)`, to send the given message on the given role.
- `SORT(Box<S>)`, to change the sort.
- `REMOVE<SkeletonLoc<S>>`, to remove a skeleton location from the skeleton graph[30].

Also, to evaluate the `seval`$(\cdot)$ predicate, in order to select a complete subset of transitions, we only need to consider transition items that correspond to message exchanges, in order to rebuild the label of

[28] Notice that the type `M` possibly contains references. This is taken into account by the Rust typesystem and even those cases are safe.

[29] In Rust terms, messages can implement the traits `Clone` or `Copy`.

[30] This transition item is not directly controlled by the user, instead, it is used internally when deallocating the location to enforce it suppression from the global location graph.

```rust
pub fn send(&mut self, r: R, msg: M) {
    if let Some(d) = self.get_endpoint(r.clone()).map(|e| e.direction()) {
        if self.t
            .last_mut()
            .is_none() {
                self.t.push(Transition::new());
            };
        self.t.last_mut().unwrap()
            .add_item(TransitionItem::SEND(r, Box::new(msg), d))
    }
}
```

Figure 5.9: The method `send` of the structure `Location` of the Rust library. This method, as well as the other methods to build transitions, works in two steps: first, if the current list of transition is empty, it creates a new empty transition; and then, it adds a `SEND` transition item in the last transition of the list.

a transition.

Therefore, we sometime use a second structure: `Label`s, which basically contains only the `SEND`, `RECEIVE`, and `EXPECT` items of transitions.

Also, according to the abstract machine, transitions need to be identified by a counter. Therefore, we also sometime use `IdentifiedLabel`, which are basically a tuple of an integer and a label.

To ease the explanation of the framework, we do not distinguish `Label`s and `IdentifiedLabel`s in the following: we will explain the behaviour the library using only `Transition`s. If one look into the source code and see a `Label`, one can understand it as a transition, and `IdentifiedLabel`s can be understood as indexed `Transition`s.

Building transitions. In the abstract machine, a location handler $\langle s, t, e, r\rangle$ accumulates transition items in t with the rules (Send), (Expect), etc..

The `Location` structure of the Rust library contains a field `t` which corresponds to the field t of the location above (See Figure 5.7).

In the Rust implementation (except for location removal, see below) the location structure implements a method for each transition item (see, for instance, the method `send` in Figure 5.9), which simply adds a new transition item in the transition being built.

If a skeleton location is required (e.g. to implement the (End) rule which adds a `Remove`$(\cdot)$ transition item to the transition being built), it is retrieved from the current location.

The case of location removal. To avoid inconsistencies such as having a location allocated, but considered *removed* by the location system, or being deallocated without being removed from the location system, location are not explicitly removable.

Ⓡ *Remark.* The Rust typesystem allows to statically compute object deallocation. Therefore, there is no garbage collection at runtime, yet there is no explicit deallocation neither.

An implicit call to the `fn drop(&mut self)` method of the `Drop` trait is automatically performed before deallocation[31].

This method is similar to destructor methods found in other languages, except that it is not explicitly called. ◁

We use the `Drop` trait to perform the removal when the location object is about to be deallocated. The implementation of `Drop` for `Location`s is given in Figure 5.10. From a programming standpoint, the main implication is that dropping a location takes time.

Committing the transaction. When all transitions are ready, the user calls `commit_changes`. This method takes over and propose the transitions to the rest of the system and wait until it is accepted or rejected. Then, it returns the result of each of the transactions[32].

Before we explain in detail how this method works, we need to have a look at the internal details of the library. This method is studied below, in Section 5.4.4.

[31] See `https://doc.rust-lang.org/std/ops/trait.Drop.html` and Appendix A.

[32] At most one is accepted and fired, but the other may be unselected for various reasons: because they are not allowed, or simply because an other was selected, etc.

```rust
fn drop(&mut self) {
    // Ignore all previous transitions item pushed.
    self.t = Vec::new();
    self.remove();
    while self.commit_changes()
        .iter()
        .filter(|res| res.is_ok())
        .next()
        .is_none() {}
}
```

Figure 5.10: The structure `Location` implements the trait `Drop`. The method `drop` resets the current transition, adds a removal token, and tries to commit the transition.

```rust
pub struct SkeletonLoc<S, R>
where
    R: RoleID,
{
    sort: S,
    prov: HashSet<R>,
    req: HashSet<R>,
}
```

Figure 5.11: Declaration of the `SkeletonLoc` structure.

5.4.2 Skeleton graphs

The module `skeletons` exports structures that implement skeleton locations and skeleton graphs.

These structures are public to the user, since it is an argument of the authorisation function.

During a location graph run, a single instance of `SkeletonGraph` is instanciated and it is shared among all locations.

Skeleton locations. Skeleton locations (structure `SkeletonLoc`, Figure 5.11) are a direct implementation of their theoretical counterparts: they simply record a sort and to sets of role identifiers. `SkeletonLoc` implements methods to apply a transition.

Skeleton graphs. Skeleton graphs (structure `SkeletonGraph`, Figure 5.12) are a set of `SkeletonLoc`. The main interest of this structure is that it implements a method `apply_transition` which applies a `Transition` to a given `SkeletonLoc` of the skeleton graph. This method is called for each transition which is selected.

Finding the skeleton location that correspond to a location. To change a skeleton location (e.g. when we apply a transition), we first need to find the right skeleton location.

If the location contains some bound roles, this task is easy, since bound roles identify locations. Otherwise, it is possible to have two skeleton locations that correspond to a single location (if multiple locations have the same sort and no bound roles).

In this case, we can actually take any of the suitable skeleton location. Therefore, when we search for a skeleton location, we take the first suitable one we find (see Figure 5.13).

```rust
pub struct SkeletonGraph<S, R>
where
    S: Clone + Eq,
    R: RoleID,
{
    locs: Vec<SkeletonLoc<S, R>>,
}
```

Figure 5.12: Declaration of the `SkeletonGraph` structure. A `SkeletonGraph` is just a list of `SkeletonLocs`.

```rust
fn search_skeleton_loc(
    &mut self,
    template: SkeletonLoc<S, R>,
) -> Option<&mut SkeletonLoc<S, R>> {
    self.locs.iter_mut().filter(|sk| template == **sk).next()
}
```

Figure 5.13: The method `search_skeleton_loc` which finds a skeleton location in a skeleton graph which suits a provided template. Notice that the method iterates over the skeleton locations and filters them, and then takes the first one (the call to `next`).

Also, notice that we return a mutable reference, i.e. the skeleton location is not removed from the skeleton graph.

When a transition is taken. Suppose we are given a set of transitions (for instance a set of transitions that have been selected for being run), it can be easily applied to the skeleton graph to get an updated skeleton graph. Each transition of the set can be applied independently: when a `Location` `l` takes a `Transition` `t`, we search for the `SkeletonLoc` that corresponds to `l`, and, for each item that has an effect on that location, we modify the `SkeletonLoc` accordingly. If `CREATE(_)` items appear in `t`, we simply add a new `SkeletonLoc` to the `SkeletonGraph` (see Figure 5.14).

5.4.3 Roles management and message exchanges

To resolve transition items `SEND(RoleID, Box<M>, RoleDirection)` and `RECEIVE(RoleID, RoleDirection)`, the library uses the `mpsc` channels of the standard library (see Appendix A). Channels are embedded into `Endpoints`, which encompass two channels (one for each direction) and a role identifier. The user has only access to role identifiers. In this section, we explain how roles are managed and used.

Roles endpoints. Locations manipulate role endpoints (see the field `roles` of the structure `Location` in Figure 5.7). A role endpoint (structure `RoleEndpoint`, see 5.15) is a simple structure that records the identifier (the field `id`) and the direction being manipulated (the field `d`), as well as two channel endpoints (the fields `s` and `r`), one to send messages, and the other to receive messages.

Remark. A `Box<V>` is a special pointer (to a value of type `V`) from the standard library. As opposed to a `&V` pointer, a `Box<V>` is allocated on the heap.

In C, both would be represented as a pointer, but a `Box<V>` corresponds to a `malloc`-ed area, while a `&V` correspond to pointer to a local variable[33]. ◁

The identifier (field `id`) is generic. This allows the user to define its own role identifiers, if they want to piggyback informations. The `RoleEndpoint` type is not visible outside the crate.

Role management. The abstract machine hides some implementation details. One of them is role management. As we have seen previously, locations have a set of `RoleEndpoint`s, each of them contains the `Sender` and the `Receiver` to exchange messages. This paragraph explains the mechanism which allows locations to resolve `Bind` and `Release` transition items.

We have seen previously that location work with `RoleEndpoint`s only, we call `Role` the structure that gathers both endpoints of a role, together with the corresponding `RoleID` (see Figure 5.16).

Locations have access, via their `LocationSystem` to a shared `RoleManager` (Figure 5.17), which collects and distribute `RoleEndpoint`s for the whole instance. Fundamentally, a `RoleManager` is just a set of `Role`s. Therefore, it contains a single field `available_roles` which has type `Vec<Role<M, R>>`.

The two roles of the `RoleManager` is to provide and get back endpoints upon request. Therefore, it offers two methods: `get_endpoint` and `release_endpoint`.

The method `get_endpoint` search for the suitable endpoint in the `available_roles` field. If not found, the role is created[34]. If found, the function returns the requested endpoint. Notice that the role contains `Option`s of `RoleEndpoint`s. If the role is already bound[35], the corresponding `Option` in the `Role` is `None`, which is returned.

The method `release_endpoint` is naively implemented[36]. Given a `r: RoleEndpoint`, it searches the

[33] This presentation is simplified, for instance, a pointer to a local variable of a calling function (i.e., in a lower stack frame) is still a `&V` in Rust (it points to a variable on the stack).

[34] Therefore, roles are created upon request.

[35] In the provided implementation, this case can not happen. To take a transition, the `Sevaluator` checks that all `Bind` transition items bind free endpoints.

[36] The function panics if we try to release a `RoleEndpoint` that was not created.

```rust
pub fn apply_transition<A, M, T>(&mut self, t: &Transition<S, M, R>, l: &Location<S
    ↪ , A, M, R, T>)
where
    A: AuthorisationFunction<Sort = S, Msg = M, RoleID = R, TransitionSet = T>,
    M: Send + Eq + Clone,
    T: TransitionSet<Sort = S, AF = A, Msg = M, RoleID = R>,
{
    self.search_skeleton_loc(SkeletonLoc::from(l))
        .expect("Couldn't␣find␣the␣skeleton␣location␣corresponding␣to␣the␣provided␣
    ↪ location.")
        .apply_transition(t);

    // Effects on the graph should be echoed *after* local effects (in case of
    ↪ removal).
    for item in t {
        self.apply_transition_item(item);
    }
}

fn apply_transition_item<M>(&mut self, item: &TransitionItem<S, M, R>)
where
    M: Send,
{
    match item {
        TransitionItem::CREATE(s) => self.locs.push(SkeletonLoc::new(*s.clone())),
        TransitionItem::REMOVE(s) => {
            let mut index = self.locs.len();
            // Notice we can not use ‘retain‘ here because there might be multiple
    ↪ skeleton
            // locations corresponding to the given one.
            for i in 0..index {
                if self.locs[i] == *s {
                    index = i;
                    break;
                }
            }
            self.locs.remove(index);
        }
        _ => (),
    }
}
```

Figure 5.14: The two methods `apply_transition` and `apply_transition_items` of `SkeletonGraph`. The method `apply_transition` of `SkeletonLoc` (not shown) naively echoes the effects of a transition on a `SkeletonLoc`.

```rust
pub (in super) struct RoleEndpoint<M, R> {
    pub id: R,
    pub d: RoleDirection,
    s: Sender<Box<M>>,
    r: Receiver<Box<M>>,
}
```

Figure 5.15: The structure `RoleEndpoint`.

```rust
struct Role<M: Send, R: RoleID> {
    id: R,
    provided: Option<Box<RoleEndpoint<M, R>>>, // Endpoint of the location that ‘provides‘ the role
    required: Option<Box<RoleEndpoint<M, R>>>, // Endpoint of the location that ‘requires‘ the role
}
```

Figure 5.16: Declaration of the `Role` structure.

```rust
pub (in super) struct RoleManager<M: Send, R: RoleID> {
    available_roles: Vec<Role<M, R>>,
}
```

Figure 5.17: Declaration of the `RoleManager` structure.

corresponding `Role` in `available_roles`, and replaces the `None` by `Some(r)`[37].

5.4.4 Transition selection and resolution

In this section, we explore how transitions are taken. The main structure used to take a transition is the `Sevaluator`. It is the structure that interacts with shared memory areas in order to synchronise transitions of different locations.

In this section, we will follow a transition, from the point it is committed by its location to the point it is resolved.

Transition authorisation. Before any attempt to enqueue a set of transitions (be it a normal attempt or a retry for a postponed transition), the `Sevaluator` checks for which transitions of the set the authorisation function holds. All transitions that do not verify the authorisation function are put aside, only the correct ones are considered. The behaviour of the authorisation function is presented in a separate section below.

Selection of a complete subset of pending transitions. Locations can submit multiple transitions at once and the system has to find which one will be taken.

In the abstract machine, the rule (SELECT) of the abstract machine selects a complete set s of transitions among the set p of submitted lists of transitions. However, the predicate $\text{pick}(\cdot,\cdot)$ does not give an actual way to *find* such a set, it only verifies that s is consistent with p; our implementation, on the other hand, has to find such a set: it can not guess the set and verify the consistency.

We present this process of selection in three steps: (i) we explain how to find if a set of transitions is complete; (ii) we show how we can keep a set of *independent* sets of transitions; and (iii) we present how we can explore all possible combinations of transitions.

We have a structure `LabelCombination` which represent a set of (identified) transitions[38], together with a *summary* of that set of transitions, i.e. the transition items that are not matched.

> **Example 17.** A `LabelCombination` can be presented as a tuple $\langle l, s\rangle$ where l is a set of labels and s is the summary, i.e. a label. For instance, if the combination contains two transitions $t_1 = \{\texttt{Send}(r_1, m_1, \texttt{provided}), \texttt{Send}(r_2, m_2, \texttt{provided})\}$ and $t_2 = \{\texttt{Receive}(r_1, \texttt{required})\}$, we can see that one of the two message in t_1 is matched by t_2 but not the other.
> In that case the corresponding `LabelCombination` would be $\langle\{t_1, t_2\}, \{\texttt{Send}(r_2, m_2, \texttt{provided})\}\rangle$, and the *summary* contains the item $\texttt{Send}(r_2, m_2, \texttt{provided})$.

The important point of this structure is that it is easy to add new transitions to the combination, while maintaining the summary correct: if we have a combination $\langle\{t_1, \ldots, t_i\}, s\rangle$ and we want to take

[37] In case we try to release an endpoint that was free (which is not possible in the implementation, since there is no double binds), the corresponding endpoint in the `Role` is not `None` but `Some(_)`, which is nonetheless replaced by the new `Some(r)`.

[38] In the implementation, we introduce `Label`s in addition to `Transition`s. A `Label` contains only the elements of the transition that enforce synchronisation, i.e. the equivalent of $\texttt{Send}(\cdot,\cdot,\cdot)$, $\texttt{Expect}(\cdot,\cdot,\cdot)$, $\texttt{Receive}(\cdot,\cdot)$, $\texttt{Bind}(\cdot,\cdot,\cdot)$ and $\texttt{Release}(\cdot\cdot,\cdot,)$ tokens. For the sake of fidelity w.r.t. the actual implementation, we distinguish them in this document, but one can think of `Label`s as `Transition`s. Notice that we define an addition on labels, which removes matching items. The type `IdentifiedLabel<R, M>` is an alias for `(Label<R, M>, u64)`, where the integer is an identifier.

```rust
struct LabelCombination<R, M>
where
    R: RoleID,
    M: Eq + Clone,
{
    labels: Vec<IdentifiedLabel<R, M>>,
    synthesis: Label<R, M>,
}
```

Figure 5.18: The structure `LabelCombination`.

```rust
impl<R, M> AddAssign<IdentifiedLabel<R, M>> for LabelCombination<R, M>
where
    R: RoleID,
    M: Eq + Clone,
{
    fn add_assign(&mut self, rhs: IdentifiedLabel<R, M>) {
        self.synthesis += rhs.0.clone();
        self.labels.push(rhs);
    }
}
```

Figure 5.19: The structure `LabelCombination` implements the trait `AddAssign<IdentifiedLabel<R, M>>`, which adds a new label into the combination. The trait `AddAssign` is used to define the `+=` operator and allows to write expressions such as `combination += label`. The trait is generic, and the generic type is the type used for the right-hand side of the operator (here, a label).

an additional transition t into account, then the combination becomes $\{t, t_1, \ldots, t_i\}, s \oplus t$ (using the definitions of the abstract machine), and similarly, to merge two combinations $\langle l_1, s_1 \rangle$ and $\langle l_2, s_2 \rangle$, the result is $\langle l_1 \cup l_2, s_1 \oplus s_2 \rangle$. The `LabelCombination` structure defines those two operations (See Figure 5.19 and Figure 5.20).

Such a structure provides a direct way to check whether the set of transitions it contains is complete: we simply have to check the `summary`, if it does not contains `SEND(_, _)`, `RECEIVE(_, _)` or `EXPECT(_, _)`, then the transition is complete.

Similarly, it is easy to check for binding issues: the `summary` field may contain `BIND(_, _)` and `RELEASE(_, _)` items. From the current skeleton graph, it is possible to create what we call a *label equivalent to the graph* (by convention, we store such label in a variable `sk_eq_label`), i.e. a fake label which contains only `BIND(r, d)` items, for each role `r` bound in direction `d` in the current skeleton graph. Since our definition of label addition removes matching `BIND`s and `RELEASE`s, then if `summary + sk_eq_label` contains a `RELEASE` item, then there is a location that tries to release an unbound role[39];

[39] Note that this may also happen if no `RELEASE` are in the sum, but there are other ways to detect such errors.

```rust
impl<R, M> AddAssign<LabelCombination<R, M>> for LabelCombination<R, M>
where
    R: RoleID,
    M: Eq + Clone,
{
    fn add_assign(&mut self, mut rhs: LabelCombination<R, M>) {
        self.labels.append(&mut rhs.labels);
        self.synthesis += rhs.synthesis;
    }
}
```

Figure 5.20: The structure `LabelCombination` implements the trait `AddAssign<LabelCombination<R, M>>`, which merges two combinations.

```rust
struct IndependentLabelSet<R, M>
where
    R: RoleID,
    M: Eq + Clone,
{
    subsets: Vec<LabelCombination<R, M>>,
}
```

Figure 5.21: Declaration of `IndependentLabelSet`, which implements a set of (independent) subsets of transitions, using a `Vec<LabelCombination>`.

and, more importantly, if two (or more) identical `BIND(r, d)` items are found, then *necessarily*, there is a location that attempts to bind a role already bound. The latter point is important, because the presence of a double `BIND` is a *necessary* and *sufficient* condition to detect such errors, and postpone the candidate label combination[40]

Now that we know how to keep sets of transitions, and how to evaluate whether such set is complete or not, we look for a way to find an complete subset out of a set of transitions. The intuition is the following: if two transitions are related (e.g. one is sending on a role, and the other is receiving on that role), then, if one is in the subset we (will) choose, then the other must also be in that subset; therefore, we can keep those two transitions together. Based on that idea, we want to keep together all related transitions, and keep separate unrelated transitions[41]. Finally, if we have a set of independent subsets of transitions, adding a new transition is easy: if the new transition is independent from all current transitions, we create a new subset, otherwise, we put it in the subset it depends on, and if the new transition is related to two (previously unrelated) subsets, we merge those two subsets.

We introduce a structure called `IndependentLabelSet` (Figure 5.21). This structure implements a set of `LabelCombination` presented above.

This structure implements a method `add_label` (see Figure 5.22), which adds a new transitions into the current ones.

Remark. Notice that transitions are proposed by locations one after the other. Therefore, we do not need to *initialise* the `IndependentLabelSet`: our `add_label` method preserves the property we are intereted in, and we begins with an empty set of subsets of transitions. ◁

Notice that, so far, we only considered that each location proposes a unique transition. In practice, we want to let a location propose multiple transtions at once, and we choose which one is actually fired. Therefore, we have to compute all possible combinations of transitions (e.g. if a location l^1 propose transitions t_1^1 and t_2^1 and a location l^2 the transitions t_1^2 and t_2^2, the combinations are $\{t_1^1, t_1^2\}$, $\{t_2^1, t_1^2\}$, $\{t_1^1, t_2^2\}$ and $\{t_2^1, t_2^2\}$). Then, for each of these combination, we can apply what has been described above to identify a potential combination that contains a complete subset.

The good news is that, as with the two previous points, given a set of combinations, it is really easy to add a new set of transitions submitted by a location: for each existing combination C_i, for each transition t_j, we create a new combination $C_i \cup \{t_j\}$[42].

This scheme is actually quite independent from the fact that we deal with transitions: more generally, suppose we have elements from two different sets $\mathbb{S}$ and $\mathbb{T}$ with an operation $\star : \mathbb{S} \times \mathbb{T} \to \mathbb{S}$, what we are doing is defining an operation $* : \mathcal{P}(\mathbb{S}) \times \mathcal{P}(\mathbb{T}) \to \mathcal{P}(\mathbb{S})$ as $S * T \triangleq \{s_i \star t_i | s_i \in S \wedge t_i \in T\}$, for $S \subset \mathbb{S}$ and $T \subset \mathbb{T}$.

We define a structure `CombinationSelector` (see Figure 5.23) which we use to compute all combinations of transitions, and which (mainly) implements two operations: the first one, shown in Figure 5.24, is the one described above (which we implement as a multiplication using the trait `MulAssign` so that we can write `combinations *= transitions`[43]), and the second is simply a filter which, given a predicate, returns a combination that satisfies the predicate, if any (this filter is later used to find a combination that contains a complete subset).

[40] We can not, however, simply reject the candidate: a future label addition might release a bound role, and make the combination acceptable.

[41] Actually, the subset we may want to choose *can* contain unrelated transitions. However, that would mean delaying the choice until two subsets are complete.

[42] Notice, however, the exponential explosion, of course.

[43] We use the `MulAssign` trait although our operation is not, strictly speaking, a multiplication.

```rust
fn add_label(&mut self, l: IdentifiedLabel<R, M>) {
    let mut independents: Vec<LabelCombination<R, M>> = Vec::new();
    let mut dependents: Vec<LabelCombination<R, M>> = Vec::new();

    for combination in self.subsets.iter_mut() {
        let mut is_independent = true;
        for (label, _) in combination.into_iter() {
            if !label.is_independent(&l.0) {
                is_independent = false;
            }
        }

        if is_independent {
            independents.push(combination.clone());
        } else {
            dependents.push(combination.clone());
        }
    }

    let merged_dependent = dependents
        .into_iter()
        .fold(LabelCombination::new(vec![l]), |acc, combination| acc + combination)
    ↪ ;

    independents.push(merged_dependent);

    self.subsets = independents;
}
```

Figure 5.22: The `add_label` method of the `IndependentLabelSet` structure. Given a new transition (the `IdentifiedLabel` `l`), we separate the current subsets into two groups, the group of subsets that do *not* contain transition related to `l` (stored in `independent`), and the group of subsets that *do* contain at least one transition related to the new one (stored in `dependent`). Once separated, we merge all subsets stored in `dependent`, and we add the new transition.

```rust
struct CombinationSelector<C>
{
    combinations: Vec<C>,
}
```

Figure 5.23: The structure `CombinationSelector`, which simply contains a set of elements of type `C`.

```rust
impl <C, E, F, I> MulAssign<F> for CombinationSelector<C>
where
    C: Add<E, Output=C> + Clone,
    F: IntoIterator<Item=E, IntoIter=I>,
    I: Iterator<Item=E>,
    E: Clone,
{
    fn mul_assign(&mut self, rhs: F) {
        let mut new_combinations = Vec::new();
        for e in rhs {
            for c in &self.combinations {
                new_combinations.push(c.clone()+e.clone())
            }
        }

        self.combinations = new_combinations;
    }
}
```

Figure 5.24: The method `mul_assign`, which is used to add a new set of transitions to a set of possible combinations. Notice the constraints put on the generic types used: we require a type `E` (that must be clonable) and that the type `C` (of combinations) implements an addition with elements of type `E` and returns a element of type `C` (our operation $\star$ above). Finally, our multiplication is defined with the right-hand side being any type `F` from which we can get an iterator over elements of type `E`.

Transition resolution. In the abstract machine, if we look at the local semantics, a transition is taken by firing the rule (Commit), the global semantics constraints whether the result is a success or a failure.

In the Rust implementation, the same mechanism appears: the method `commit_changes` (Figure 5.25) attempts to commit a list of transition, and returns a list of results (among which at most one is a success).

This method proceeds in two steps: first, the list of transitions is given to the `Sevaluator`, which decides whether one of the transitions is accepted or not, according to the previous paragraph (this corresponds to the rule (Select) of the global semantics). This is performed using the method `add_label` of the `Sevaluator` structure.

Remark. The method `add_label` is blocking. It loops until a result (either a success or a failure) for the submitted list is available, then it returns it. ◁

Remark. Notice the while loop which transparently retries the transition until `add_label` returns only `Err(SevaluatorError::NotSelected)` or `Err(SevaluatorError::RetryPostponed)`.

`Err(SevaluatorError::RetryPostponed)` happens when an other transition is selected, but the current one can not be rejected neither. This correspond to the rule (Retry) of the abstract machine. ◁

Then, the `SevaluatorResults` are converted in `TransitionResults`: for error, the conversion is simply a mapping, and if a transition among the proposed one is selected, then it is *resolved*[44]; and the results of that transition are then returned.

Let now see how that selected transition is resolved. We simply apply each of the `TransitionItem`s that compose the transition, using a method `apply_transition_item` in a loop.

We will only look at message sending (receiving is quite similar) and role binding (releasing is quite similar). Location creation and changing the sort are intuitively implemented, location removal should be intuitive after reading the paragraph on removal in Section 5.4.1, and *expect* transition items are already checked by the `Sevaluator`, and therefore do not need to be checked again.

From the selection, we know that, for each role, each `SEND(r, m, d)` is matched by a `RECEIVE(r, d)`. To send a message, we first get the endpoint of the location from its field `roles`, and from the right endpoint, we simply send the message using the underlying channel (see Figure 5.26).

To bind (or release) a role, we interact with the role manager as explained above: we request a new role, and we add it to the endpoints of the location (see Figure 5.27).

[44] See the loop in which the method `apply_transition_item` is called.

```rust
pub fn commit_changes(&mut self) -> Vec<Result<Vec<TransitionResult<S, A, M, R, T
    ↪ >>, TransitionError>> {
    let mut seval = self.ls.seval.clone().add_label(&self.t, &self);

    // While this label is postponed, check authorisation and retry.
    while seval.iter().fold(true, |acc, res|
        acc && (*res == Err(SevaluatorError::RetryPostponed)
            || *res == Err(SevaluatorError::NotSelected))
    ) {
        seval = self.ls.seval.clone().add_label(&self.t, self);
    }

    let mut transitions: Vec<Transition<_, _, _>> = self.t.drain(..).collect();
    for t in &mut transitions {
        t.sort();
    }

    let zipped = seval.iter().zip(transitions);
    let result_per_transition = zipped.map(|(res, t)|
        match res {
            Err(SevaluatorError::NotAuthorised) => Err(TransitionError::
    ↪ NotAuthorised),
            Err(SevaluatorError::ExpectedNoMatch) => Err(TransitionError::
    ↪ ExpectedNoMatch),
            Err(SevaluatorError::BindNotAvailable) => Err(TransitionError::
    ↪ BindNotAvailable),
            Err(SevaluatorError::NotSelected) => Err(TransitionError::NotSelected),
            Err(SevaluatorError::NotInTransSet) => Err(TransitionError::
    ↪ NotInTransSet),
            Err(SevaluatorError::RetryPostponed) => {
                panic!("Found␣SevaluatorError::RetryPostponed␣outside␣the␣retry␣
    ↪ loop.")
            }
            Ok(_) => {
                let mut ret = Vec::new();
                for item in t {
                    if let Some(result) = self.apply_transition_item(item) {
                        ret.push(result)
                    }
                }
                Ok(ret)
            }
        }
    );
    result_per_transition.collect()
}
```

Figure 5.25: The method `commit_changes` of the structure `Location` of the Rust library.

```
TransitionItem::SEND(r, msg, _) => {
    if let Some(ep) = self.get_endpoint(r) {
        ep.get_sender()
            .send(msg)
            .expect(&"Could not send.");
    } else {
        panic!("Try to send to a role that is not bound.")
    }
}
```

Figure 5.26: Extract of the method `apply_transition_item`. This extract correspond to the application of a `TransitionItemType::SEND(r, m, d)` transition item.

```
TransitionItem::BIND(r, d) => {
    if let Some(ep) = self.ls.rm.lock().unwrap().get_endpoint(r, d) {
        self.roles.push(ep)
    } else {
        panic!("Try to bind a role which is not available.")
    }
}
```

Figure 5.27: Extract of the method `apply_transition_item`. This extract correspond to the application of a `TransitionItemType::BIND(r, d)` transition item.

5.4.5 Authorisation functions and unconstrained location transitions set

In the presentation above, we mentioned the authorisation function without explaining it. In this last section, we present it. It is implemented as a simple trait to be implemented by the user.

Authorisation functions. Our goal is to let the user provide an authorisation function. This authorisation function should take a location (i.e. an instance of `Location`), a skeleton graph (i.e. an instance of `SkeletonGraph`), and a transition (i.e. an instance of `Transition`).

We define a trait `AuthorisationFunction` (see Figure 5.28) which contains a function `authorise` which shall take the argument presented above, and shall return a boolean (i.e. a `bool`). Notice that, in order to prevent the user to modify any of the provided argument (typically, we do not want to allow it to modify the skeleton location), the `authorise` function actually takes (immutable) references to the mentioned arguments.

Finally, notice that the trait requires its implementations to be clonable[45], so that each (local) instance of `Sevaluator` has a copy of the authorisation function.

Unconstrained location transitions set. Similarly, we define a trait `TransitionSet` (Figure 5.29) which defines a method `contains`, used to state whether a given transition is in the set of unconstrained location transitions of the instance.

This is quite similar to the `AuthorisationFunction` trait, except that, as in the theoretical definition, `contains` does *not* depend on the skeleton graph. Therefore, it can be evaluated without locking the shared variables.

5.5 Utilities

While implementing various examples using the `rust_locations` framework, we had to develop some pieces of code that could be useful in other location graph instances. Although these pieces do not extend the expressive power of the library[46], we found useful to extract those pieces in a `rust_locations::utils` module.

In this section, we present those utilities. First, we will present an implementation of a trivial authorisation function and a trivial set of unconstrained location transitions. Those two structures implement respectively an authorisation function and a set of unconstrained location transitions without constraints.

[45] A clone is a deep copy, see the Appendix A.

[46] These pieces of code do not require any access to internal details of the library, and are therefore implementable by any user using only the provided API.

```rust
pub trait AuthorisationFunction: Sized + Clone {
    /// The type of sorts for this instance of location graphs.
    type Sort: Clone + Eq;
    /// The type of messages that are exchanged in this instance.
    type Msg: Send + Eq + Clone;
    /// The type of roles names.
    type RoleID: RoleID;
    /// The type of the unconstrained location transition set.
    type TransitionSet: TransitionSet<Sort=Self::Sort, Msg = Self::Msg, RoleID =
    ↪ Self::RoleID, AF = Self>;

    /// Returns true if the transition (i.e. the set of message exchange, role
    ↪ binds/releases and
    /// sort changes) is allowed.
    ///
    /// The parameter `t` is the `Transition` to be allowed; `s` is the `
    ↪ SkeletonGraph` of the
    /// location graph at the instant the transition is taken; and `l` is the
    ↪ location that takes
    /// the transition.
    ///
    /// Notice that, contrarily to the paper, the authorisation function is *
    ↪ stateful*, i.e. the
    /// authorisation function takes a *mutable* reference to `self`.
    ///
    /// The default implementation is always true.
    // The arguments are not used in the default implementation, but may be used in
    ↪  dedicated
    // implementations.
    #[allow(unused_variables)]
    fn authorise(
        &mut self,
        s: &SkeletonGraph<Self::Sort, Self::RoleID>,
        t: &Transition<Self::Sort, Self::Msg, Self::RoleID>,
        l: &Location<Self::Sort, Self, Self::Msg, Self::RoleID, Self::TransitionSet
    ↪ >,
    ) -> bool {
        true
    }
}
```

Figure 5.28: The definition of the trait `AuthorisationFunction`. Notice that a default implementation is provided, which is always true.

```rust
pub trait TransitionSet: Sized + Clone {
    /// The type of sorts for this instance of location graphs.
    type Sort: Clone + Eq;
    /// The type of messages that are exchanged in this instance.
    type Msg: Send + Eq + Clone;
    /// The type of roles names.
    type RoleID: RoleID;
    /// The authorisation function used for this location system.
    type AF: AuthorisationFunction<Sort=Self::Sort, Msg=Self::Msg, RoleID=Self::
    ↪ RoleID, TransitionSet=Self>;

    /// Returns `true` if the given transition is in the set of unconstrained
    ↪ transition
    /// transitions.
    ///
    /// It may seem similar to the `AuthorisationFunction` trait. Actually, it is
    ↪ possible to
    /// verify in the authorisation function that the transition is correct.
    ↪ However, one should
    /// prefer to use this trait as much as possible, for performance reasons:
    ↪ checking the
    /// authorisation function requires to have an view of the location graph,
    ↪ which have to be
    /// atomic with respect to the effect of the transition, this requires, at some
    ↪  point, locking
    /// a mutex on the location graph.
    ///
    /// Since evaluating whether a transition is in the set of unconstrained
    ↪ location transitions,
    /// there is no need to perform this global lock, and this test can be
    ↪ performed only once even
    /// if the transition is tried multiple times.
    ///
    /// The default implementation is always true.
    // The arguments are not used in the default implementation, but may be used in
    ↪  dedicated
    // implementations.
    #[allow(unused_variables)]
    fn contains(
        &self,
        t: &Transition<Self::Sort, Self::Msg, Self::RoleID>,
        l: &Location<Self::Sort, Self::AF, Self::Msg, Self::RoleID, Self>,
    ) -> bool {
        true
    }
}
```

Figure 5.29: The definition of the trait `TransitionSet`. This trait requires a method `contains`, used to indicate which transitions are in the set of unconstrained location transition of the model. A default implementation includes all transitions.

The second utility we introduce is a structure for generic role names. While some systems require a particular structure for role names (for instance, to implement ownership-based systems, where sorts are used as role names to ensure uniqueness), we often do not care about the structure of those names. Therefore, we implemented the `DefaultRole` structure, which is a blackbox structure for role names.

Then, we introduce locations to interact with TCP sockets. From the examples we will detail below, we realised that interacting via TCP would be useful in a lot of cases. We therefore implemented such locations and introduced them in the crate.

5.5.1 Trivial authorisation function and transition set

In this section, we present the `TrivialAF` and `TrivialTS` structures, which respectively implement the trivial authorisation function and the trivial set of unconstrained location transitions.

In addition, in a second paragraph, we also present three helper functions that can be used to ease writing authorisation functions.

Trivial authorisation function and set of unconstrained location transitions. If, for any reason, the user does not need to use either the authorisation function or if they want to have the biggest set of unconstrained location transitions, then the `AuthorisationFunction` and the `TransitionSet` can be trivially implemented. Since any of both situation happens quite often, we provide those trivial implementations.

The `AuthorisationFunction` trait already provides a default implementation for `authorise`, therefore, we just have to implement a dummy structure which we call `TrivialAF` (resp. for `TrivialTS`). Figure 5.30 shows this trivial implementation.

Remark. The only subtlety of this implementation comes from Rust. The compiler does not accept generic types for a structure, if those generic types do not appear in the types of fields of the structure. In our case, our structure does not need any field, and this would prevent us from using generic types.

Fortunately, Rust provides a special structure `PhantomData<T>`, which is a zero-sized structure used to fake fields.

In our case, the `TrivialAF` structure is generic over four types (the four associated types of the `AuthorisationFunction` trait), and therefore contains four `PhantomData` fields. ◁

The traits `AuthorisationFunction` and `TransitionSet` being very similar, the implementation of the trivial set of unconstrained location transitions (`TrivialTS`) is quite similar, and therefore not shown.

Helper functions. In this paragraph, we present three helper functions that can be used to write authorisation functions and transition sets. The first function is `authorise_removal`, which takes a transition and accepts it if it contains a location removal. The second function is `binds_only` which takes a transition and a predicate over roles and role directions, and ensures that all roles that are bound during the transition respect the predicate. Finally, the third function is `forbid_self_removal` which takes a location and a transition, and ensures that the transition does not removes the provided location.

First, about `authorise_removal`. We usually want to allow locations to get removed. Therefore, we might often want our authorisation functions and our transition sets to allow transitions if they contains a `TransitionItem::REMOVE(_)` item. Since this is a common pattern, we provide the function `fn authorise_removal(t: &Transition<S, M, R>)` which takes a transition and returns true in that case. This function, simply implemented by iterating over all elements of a transition and returning early when a `REMOVE` item is found, is shown in Figure 5.31.

Second, the function `binds_only`. We often want to restrict some location to bind only some role names in some direction (typically, if locations should bind their sort, we want to prevent them from binding other sorts, or if we have a main component that binds in a given direction, and its subcomponents that bind in the other direction, etc.). The function `binds_only` takes a predicate over roles and directions (an instance of `Fn(&R, &RoleDirection) -> bool`, where `R` is the type of roles), and verifies, by iterating over all transition items of a transition, that all `BIND` items respect the predicate. The implementation of this function is shown in Figure 5.32.

Finally, the function `forbid_self_removal`. When some locations should be present during the whole execution (consider for instance a location that listen for connections on a network socket), we may want to forbid their removal. This function perform that task. Following the same iterator pattern than the two previous helper functions, it checks for all transition items of a given transition that none is a `REMOVE` of a given location. The implementation is shown in Figure 5.33

```rust
#[derive(Clone, Copy)]
/// A trivial authorisation function: allow any transition.
pub struct TrivialAF<S, M, R, T> {
    s: PhantomData<S>,
    m: PhantomData<M>,
    r: PhantomData<R>,
    t: PhantomData<T>,
}

impl<S, M, R, T> TrivialAF<S, M, R, T> {
    /// Creates a new `TrivialAF`.
    pub fn new() -> Self {
        TrivialAF {
            s: PhantomData,
            m: PhantomData,
            r: PhantomData,
            t: PhantomData,
        }
    }
}

impl<S, M, R, T> AuthorisationFunction for TrivialAF<S, M, R, T>
where
    S: Clone + Eq,
    M: Send + Eq + Clone,
    R: RoleID,
    T: TransitionSet<Sort = S, AF = Self, Msg = M, RoleID = R>,
{
    type Msg = M;
    type Sort = S;
    type RoleID = R;
    type TransitionSet = T;
}
```

Figure 5.30: Implementation of the structure `TrivialAF`. The `PhantomData` is a marker to indicate that we do not use a provided generic type (it is a 0 size placeholder provided in the standard library to make the analyser happy). Notice that, in the implementation of `AuthorisationFunction`, we only need to define the associated types, as the function **`authorise`** has a default implementation.

```rust
pub fn authorise_removal<S, M: Send + Eq + Clone, R: RoleID>(t: &Transition<S, M, R
    ↪ >) -> bool {
    t.into_iter().fold(false, |acc, item| {
        if let TransitionItem::REMOVE(_) = item {
            true
        } else {
            acc
        }
    })
}
```

Figure 5.31: The helper function **`authorise_removal`** of the **`utils`** sub-module. The function takes a transition and returns true if it contains a `TransitionItemType::REMOVE` transition item.

```rust
pub fn binds_only<S, M, R>(
    t: &Transition<S, M, R>,
    p: impl Fn(&R, &RoleDirection) -> bool,
) -> bool
where
    M: Send + Eq + Clone,
    R: RoleID
{
    for t_item in t {
        if let TransitionItem::BIND(r, d) = t_item {
            if !p(r, d) {
                return false;
            }
        }
    }
    true
}
```

Figure 5.32: Implementation of the `binds_only` helper function. This function verifies that a predicate `p` over roles and directions holds for all newly bound roles.

```rust
pub fn forbid_self_removal<S, A, M, R, T>(
    t: &Transition<S, M, R>,
    l: &Location<S, A, M, R, T>,
) -> bool
where
    S: Clone + Eq,
    A: AuthorisationFunction<Sort = S, Msg = M, RoleID = R, TransitionSet = T>,
    M: Send + Eq + Clone,
    R: RoleID,
    T: TransitionSet<Sort = S, AF = A, Msg = M, RoleID = R>,
{
    t.into_iter().fold(true, |acc, item| {
        if let TransitionItem::REMOVE(s) = item {
            let sl = SkeletonLoc::from(l);
            if &sl == s {
                false
            } else {
                acc
            }
        } else {
            acc
        }
    })
}
```

Figure 5.33: Implementation of the `forbid_self_removal` helper function. This function verifies that no transition item of a transition attempt to remove a given location, based on the skeleton of that location.

```rust
#[derive(Debug, Copy, Clone, PartialEq, Eq, PartialOrd, Ord, Hash)]
pub struct DefaultRole {
    id: i32,
}

impl RoleID for DefaultRole {}

impl Display for DefaultRole {
    fn fmt(&self, f: &mut Formatter<'_>) -> std::fmt::Result {
        write!(f, "id␣{}", self.id)
    }
}
```

Figure 5.34: Implementation of the structure `DefaultRole`.

5.5.2 Generic role names

Some location graph instances require the role names to have a structure, typically to statically guarantee that some property hold. For instance, our implementation of the owners-as-dominators policy (see 4.1.1) required roles to have a special shape to guarantee the property. However, we quite often do not need such structure: role names can been plain integers, or even black boxes.

We therefore implemented the `DefaultRole` structure, which implements the `RoleID` trait and that can therefore be used as a black box type for roles of a given instance. In order to instantiate new roles, we provide a `DefaultRoleGenerator` which provides a method `new_id(&mut self) -> DefaultRole`.

Internally, the `DefaultRole` structure is just a wrapper over Rust integers (type `u32`), as shown in Figure 5.34.

Concerning the generator, we intentionally implemented this wrapper naively (see Figure 5.35): two `DefaultRoleGenerator`s can create two roles that are equals; and the only ensured guarantee is that two roles created by the same `DefaultRoleGenerator` are distinct. This behaviour is particularly useful when two distinct locations have to bind a certain amount of roles: both can have distinct instances of `DefaultRoleGenerator` and pull roles names from these instances: the roles will match.

> Implementing a generator of `DefaultRole`s such that two generators yield distinct role would be quite straightforward to implement. Two possibilities could be: (i) each generator shall be initialised with a (distinct) prime integer p and would yield the roles p^n for $n \geq 1$.; and (ii) each generator shall be initialised with a distinct integer i and would yield $\langle i, n \rangle$ for $n \geq 1$ (and two roles $\langle i_1, n_1 \rangle$ and $\langle i_2, n_2 \rangle$ are equals if and only if $i_1 = i_2 \wedge n_1 = n_2$). The first possibility is a bit more convoluted but is compatible with our current `DefaultRole`s.

5.5.3 TCP connections

General approach. In the following section (Section 5.6.2), we will show, as an example, how to implement a Publish-Subscribe server. To implement that, we needed to establish TCP connections in order to exchange with the clients. It turned out that implementing such TCP connections is not that straightforward: we have to poll the connection for any data received and, *in the meantime*, wait for some data to send, provided by some other parts of the location graph.

Both polling are blocking and, even if they were not, implementing some kind of timeout would be inefficient. We therefore took an other strategy: we implemented two kinds of locations: `TcpLocation` and `TcpReader` which wrap the TCP connection: i.e. at any time, we can either send data to the `TcpLocation`, which will manage to forward it on the TCP connection without delay[47]; or we can attempt to receive data from the `TcpReader`, and read the data received, if any[48].

While we can not poll simultaneously a role and a TCP connection, we can, however, poll two roles simultaneously: we simply attempt two transitions, and the first possible will be selected (or any of the two if both are possible).

[47] Well, there are always some delay, at least due to the connection, but the point is that the data will be sent without being blocked by an attempt to receive data from the TCP connection.

[48] If no data is to be read, the `TcpReader` does not propose the complementary *send* and, therefore, the transition can not fire.

```rust
#[derive(Copy, Clone)]
pub struct DefaultRoleGenerator {
    i: i32,
}

impl DefaultRoleGenerator {
    /// Returns a new `DefaultRoleGenerator`.
    pub fn new() -> Self {
        DefaultRoleGenerator { i: 0 }
    }

    /// Generates a new `RoleID`.
    pub fn new_id(&mut self) -> DefaultRole {
        self.next().unwrap()
    }
}

impl Iterator for DefaultRoleGenerator {
    type Item = DefaultRole;
    fn next(&mut self) -> Option<Self::Item> {
        let r = DefaultRole { id: self.i };
        self.i += 1;
        Some(r)
    }
}
```

Figure 5.35: Implementation of the generator of `DefaultRole`. Notice that it implements the `Iterator` trait almost for free.

Remark. Rust handles TCP connections with the `TcpStream` structure of the standard library. In particular, this structure implements the method `try_clone` which attempts to create an independant duplicate of the stream. Therefore, it is possible to have both a `TcpLocation` and a `TcpReader` for a single connection. ◁

The `TcpLocation` performs a simple loop in which it proposes the following transition: it receives some data from the *write* role. Upon performing that transition, the `TcpLocation` forward the data on the TCP connection and the loop starts again.

The `TcpReader` performs the opposite loop: it polls continuously the TCP connection and, upon reception of a line of data[49], attempt to send it on a role which we call the *read* role. Notice that until the data is not forwarded (i.e. until an other location receives from the *read* role), the `TcpReader` location is blocked, and further data received on the TCP connection is buffered.

From this side, the naming convention may be unintuitive: the `TcpLocation` reads from the role `self.write`, and vice-versa for `TcpReader`. We name the role from the outer world perspective: the outer world *writes* on the role *write* and *reads* from *read*.

Finally, we also add a third structure: `TcpListenerLocation` which listens for new TCP connections and, when such a new connection is established, evaluates a function.

Types for TCP locations. In order to interact with wider graphs, our two locations shall be able to interact with various kinds of messages, sorts, and roles. For sorts and roles, they can be implemented as generic types without restriction. Messages, however, need a bit more work: we need an local type for messages (which we call `TcpMessage`), which contains two variants: (i) one which contains data (a wrapper over `String`s); and (ii) the other for administrative content (currently, we only report that the connection shut down) (see Figure 5.36). When operating on a wider graph which uses a generic type `M` for messages, we require that `M` can be converted to `TcpMessage` and the other way around, using the `From` and `TryInto` traits of the standard library[50].

[49] For the sake of simplicity, we assume the data received is always an UTF8 string, which we buffer by line using `BufReader` from the standard library.

[50] Notice that we require the `TryInto` trait and not `Into`. The difference is that `TryInto` can fail, as its name suggests. We choose this option because it may be the case that the user does not want to convert all `M` into `TcpMessage`s. From a more abstract point of view, we want the conversion to be a surjective partial function.

```rust
#[derive(Debug, Clone)]
pub enum TcpMessage {
    /// A variant which contains a line of text received or to be sent.
    Data(String),
    /// A generic error. In case of failure, such message is propagated and
    ↪ locations are removed.
    Shutdown,
}
```

Figure 5.36: The type `TcpMessage` used by the `TcpLocation` and `TcpReader` locations.

```rust
pub struct TcpLocation<S, A, M, R, T>
where
    S: Clone + Eq,
    A: AuthorisationFunction<Sort = S, Msg = M, RoleID = R, TransitionSet = T>,
    M: Send + Eq + Clone + From<TcpMessage> + TryInto<TcpMessage>,
    R: RoleID,
    T: TransitionSet<Sort = S, AF = A, Msg = M, RoleID = R>,
{
    l: Location<S, A, M, R, T>,
    stream: TcpStream,
    write: R,
}
```

Figure 5.37: The declaration of `TcpLocation`. Notice the constraint `M: From<TcpMessage> + TryInto<TcpMessage>`, which means that the user of the `TcpLocation` has to implement the `From<TcpMessage>` and `TryInto<TcpMessage>` traits for `M`.

The `TcpLocation` structure. The declaration of `TcpLocation` is shown in Figure 5.37. The initialisation of that location simply consists in binding the *write* role (the field `self.write` of the structure) and creating the `TcpReader` location, and is therefore not shown here. The main loop is implemented in the method `TcpLocation::run` which is called after initialisation (see Figure 5.38). It consists in a loop in which the location attempts to receive from the role `self.write`. When the transition is performed, the result is checked and, in case of success of the transition, two cases may occurs: (i) the message can be converted into `TcpMessage::Data(s)`, which is written on the TCP connection; and (ii) the message can be converted into `TcpMessage::Shutdown`, in which case the `TcpLocation` quits. If the transition is not successful or if the conversion from `M` to `TcpMessage` fails, then the error is ignored and we proceed to the next iteration of the loop.

The `TcpReader` structure. Finally, `TcpReader` is even simpler. Its declaration is quite similar to `TcpLocation` and its initialisation consists only in binding the *read* role. Therefore, we do not show these two sections of the code. Its main loop simply reads a line from the `TcpStream`[51] and upon reading, performs a transition in which the data is sent. The result of the transition is ignored. If reading is not possible, for any reason (the main one being that the connection was shut down), then the location sends a `TcpMessage::Shutdown` to the outer world and quits. This main loop is shown in Figure 5.39.

The `TcpListenerLocation` structure. The `TcpListenerLocation` structure is quite simple: it uses the `TcpListener` structure from the standard library to continuously listen for new connections. The structure also has a field `state: St` which is a state of generic type `St` the programmer can use (for instance to increment a counter everytime a new connection is established).

The action to perform upon new connection is given as a function with type `Fn(&mut St, &mut Location<S, A, M, R, T>, TcpStream)`. Additionnaly, the user can provide an other function to perform additionnal actions at initialisation (e.g. binding some roles).

As shown in Figure 5.40, this location simply creates a `TcpListener` binding the given address. The `incoming` method returns an iterator over new connections, which is blocking when no new connection is available, which we use to call the given function `f_connect` on each new stream.

[51] To read data lines by line, the `TcpStream` is embedded in a `BufReader<TcpStream>`, which buffers the data received and allows to access it by lines.

```
self.l.receive(self.write.clone());
for transition_result in self.l.commit_changes() {
    match transition_result {
        Ok(item_results) => {
            for item_res in item_results {
                match item_res {
                    TransitionResult::RECEIVED(_, m) => {
                        match (*m).try_into() {
                            Ok(TcpMessage::Data(line)) => {
                                self.stream.write(line.as_bytes()).unwrap();
                            }
                            Ok(TcpMessage::Shutdown) => {
                                return;
                            }
                            Err(_) => {
                                eprintln!("Could␣not␣convert␣a␣message␣(type␣M)␣to␣
    ↪ a␣TcpMessage.");
                            }
                        }
                    }
                    TransitionResult::CREATED(_) => {
                        eprintln!("TcpLocation␣unexpected␣created␣location.");
                    }
                }
            }
        }
        Err(e) => {
            eprintln!("TcpLocation␣error␣while␣committing␣a␣transition:␣{:?}.", e);
        }
    }
}
```

Figure 5.38: The main loop of `TcpLocation`s. Although quite long, it is actually pretty simple: it begins by creating and performing a simple reception on `self.write`. Then, most of the lines are just here for opening the result. The innermost `match` performs the actual work: if some data is received, then it is written to the TCP connection; if a `TcpMessage::Shutdown` is received, the function returns, and the location ends. It is then removed as usual, in the `drop` method (see the paragraph on location removal in Section 5.4.1).

```
for line in reader.lines() {
    if let Ok(line) = line {
        self.l
            .send(self.read.clone(), TcpMessage::Data(line).into());
        self.l.commit_changes();
    } else {
        self.quit();
        return;
    }
}
self.quit();
```

Figure 5.39: The main loop of the `TcpReader` structure. It simply get lines from the given `TcpStream` and forward them on the role `self.read`. The method `self.quit()`, not shown, simply sends a `TcpMessage::Shutdown` on `self.read`.

```rust
pub fn run(mut self,
    f_init: impl FnOnce(&St, &mut Location<S, A, M, R, T>) -> (),
    f_connect: impl Fn(&mut St, &mut Location<S, A, M, R, T>, TcpStream) -> ()
) {
    println!("Run␣TCPListenerLocation.");
    let listener =
        TcpListener::bind(self.addr).expect("Could␣not␣bind␣the␣given␣address.");

    f_init(&self.state, &mut self.l);

    for res_stream in listener.incoming() {
        let stream = res_stream.expect("Could␣not␣establish␣stream.");
        f_connect(&mut self.state, &mut self.l, stream);
    }
}
```

Figure 5.40: The `run` method of `TcpListenerLocation`, which implements the main loop of that location. This method takes two arguments `f_init` and `f_connect` which are both functions: the former is called once before begining to listen for new connections; and the latter is called each time a new connection is established.

```
create
Created account 1
share 1 1
Successfully shared
credit 1 100
Credited
```

(a) Client 0

```
New shared account 1
withdraw 1 100
Withdraw success
```

(b) Client 1

Figure 5.41: A run of our bank system. In this run, a client (Client 0) creates an account and shares it with an other client (Client 1). Client 1 is notified of the sharing. Both clients can access the account, e.g. Client 0 credits 100 units on the account, which are withdrawn by Client 1.

5.6 Encapsulation policies in Rust using Location Graphs

In this section, we implement two examples of location graph based programs. Those two examples implement two encapsulation policies presented previously. The first example is the implementation of a bank system, such as presented in Section 1.2. The policy implemented by this system is a (slightly modified) owners-as-ombudsmen policy. The second example is an implementation of Publish-Subscribe server which internally uses a logging system such as the one presented in the introduction.

5.6.1 An application of the owners-as-ombudsmen: a bank system

Bank system. We want to create a simple bank system, in which clients can create and share accounts, and credit and withdraw units from their accounts. Clients interact with the bank using TCP connections. They are offered a few commands: (i) `create` which creates a account (the number of the account is printed back); (ii) `credit i j` which credits `j` units on account `i`; (iii) `withdraw i j` which attempt to withdraw `j` units from account `i`; and (iv) `share i j` which allow client `j` to access account `i`. A run is shown in Figure 5.41. Notice that account numbers are local to a client (i.e. Account 1 of Client 1 is not necessary the same account than Account 1 of Client 2), also a shared account does not necessarily have the same number for all clients.

Remark. For the sake of simplicity, we do not manage the addition or removal of clients: our bank contains a fixed number of clients, statically chosen. Also, the ports clients use are statically assigned. In the example presented, we only use two clients, which listen respectively on ports 10000 and 10001. ◁

General overview. To implement our system, we have three kinds of components: (i) an Bank component, which coordinates actions performed by the other components (in particular the creation of names); (ii) Client components, which perform operations on accounts according to orders received on TCP connections; and (iii) Account components, which hold the state of all accounts. As stated in the introduction (Section 1.2), Accounts can be shared among multiple Clients, and Clients should be able

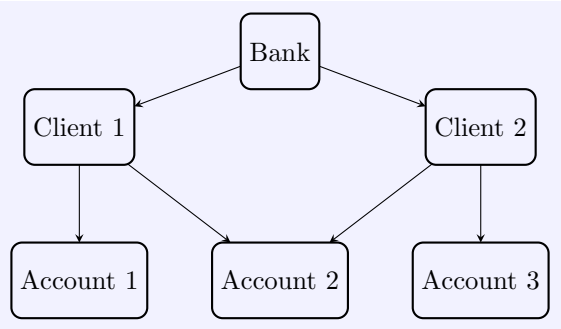


Figure 5.42: Ownership relations in an instance of the bank system. In that example, three accounts are owned by two clients: two of them are exclusive and one is shared. The Bank component is considered as an owner of the two Client components.

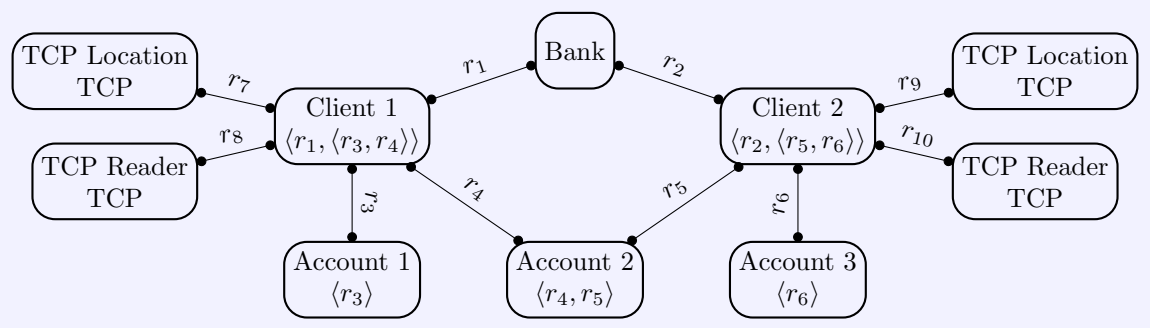


Figure 5.43: Overview of the bank system. The bank component establishes links with all clients. To create an account, a client sends the bank a message to get a new role, which is used by the client and the (newly created) account. When sharing an account, the current owner (or one of the current owner) tells the Bank to share the account with an other client. To do so, the Bank creates a new role name which it forward to both the new shared owner of the account; and the client which preforms the sharing request. The clients which performs the request forward the new role to the account, so that both the new shared owner and the account can use the new role.

to access each of their accounts (and only those). For the sake of simplicity, we keep the specification of Bank accesses very loose. It turned out, during the implementation, that the bank only needs to access the client components[52] to share role names. Therefore, we enforce the following policy, which is an slight extension of the owners-as-ombudsmen policy, in order to take into account the Bank component: except for the Bank, we have a owners-as-ombudsmen policy, where accounts are owned by one or multiple clients; the Bank component is, in addition, considered as the owner of all Client components. Figure 5.42 shows an example of ownership relations in a particular instance of the bank system.

This system is easily implemented using the location graph framework. We follow the procedure explained in Section 4.1.2: Client locations expose, in their sort, an identifier and the identifier of all Account locations they own. Account locations, on their side, expose on their sort a set of identifiers. In addition, to take into account the additional Bank location, we consider that the Bank has a special sort, and it binds all Client locations' identifiers.

In addition, not addressed in this introduction, we have TCP locations. For the sake of simplicity, all TCP related location have a special sort *TCP*. To focus on the *owners-as-ombudsmen* aspects of the example, we do not constraint their communications. Figure 5.43 shows the location graph instance which correspond to the example shown in Figure 5.42.

Types. In addition to TCP related locations[53], we have three kinds of locations: (i) `BankLocation` which implements the main Bank component; (ii) `ClientLocation` which implements individual clients, and uses additional TCP locations for interactions with the user(s); and (iii) `AccountLocation` which implements individual accounts.

[52] It is also very easy to imagine an implementation where the Bank component is not even needed, provided Clients can generate distinct role names. We did not choose this solution for the sake of simplicity.

[53] See Section 5.5.3.

```
Messages::Shared(r) => {
    let mut rid = self.l.sort().account().unwrap().clone();
    rid.push(r);
    self.l.change_sort(BankSort::Account(rid));
    self.l.bind(r, RoleDirection::REQUIRED);
    self.l.commit_changes().pop().unwrap().unwrap();
}
```

Figure 5.44: Extract of the main loop of the `AccountLocation` structure. This extract shows the actions the account performs when it receives a `Shared(r)` message from an owner. In that case, the location changes its sort and binds the new role.

We use `DefaultRole`[54] as roles.

We have four kinds of sorts: (i) `Bank`, for the Bank location; (ii) `Client(Role, Vec<Role>)`, for clients; (iii) `Account(Vec<Role>)`, for accounts; and (iv) `TCP`, for TCP related locations. Notice that both `Client(Role, Vec<Role>)` and `Account(Vec<Role>)` expose some roles, according to the implementation of owners-as-ombudsmen presented in Section 4.1.2.

Remark. For the sake of simplicity, since TCP locations and the Bank location are addition to the owners-as-ombudsmen, we do not bother constraining their accesses. ◁

Finally, messages range over ten variants: (i) `NewAccountReq`, sent from a client to the Bank upon the creation of a new account, to retrieve a fresh role; (ii) `NewAccount(Role)`, the answer from the Bank, with the fresh role piggybacked; (iii) `Credit(u32)`, sent from a client to an account, to credit that account of a given amount; (iv) `Withdraw(u32)`, sent from a client to an account, to attempt to withdraw a given amount from that account; (v) `Close` sent from a client to an account, to attempt to close that account; (vi) `Success`; (vii) `Failure`, both used as response from the account to the client; (viii) `Share(usize)`, sent from a client to the Bank, to inform the bank that the client whishes to share an account with an other client (with the given index); (ix) `Shared(Role)`, the answer from the Bank, sent to the initial client (which forwards that message to the relevant account) and the new shared owner; and (x) `Tcp(String)`, for all communications between a client and its TCP locations.

Locations. `AccountLocation`s implement Account components. The structure has two fields: (i) the underlying `Location`; and (ii) the current `amount` of units stored in the account. The sort of accounts, which is the variant `Account(Vec<Role>)` of `BankSort` keeps the set of roles to owners. Their behaviour is quite straightforward: they wait for messages incoming from one of their owner and react accordingly (if the message is one of `Credit`; `Withdraw`; `Close`; or `Shared`), or ignore them otherwise. The only non-trivial action is if a `Shared(r)` message is received, indicating that the account has a new shared owner. In that case, the location has to perform an additional transtion to bind the role `r` (the line `self.l.bind(r, RoleDirection::REQUIRED)`) and update its sort to reflect that it has a new owner (the line `rid.push(r)` adds the role name `r` to the list of role names currently in the sort, then `self.l.change_sort(BankSort::Account(rid))` updates the sort). This case is shown in Figure 5.44.

The Bank location (structure `BankLocation`) manages the creation and the sharing of accounts. In addition, it also generates all role names used in the system. That structure has three fields: (i) the underlying `Location`; (ii) a `DefaultRoleGenerator`, presented in Section 5.5.2; and (iii) a set of roles[55], named `clients`, to communicate with the clients. The behaviour of the Bank location is simply to listen to all clients (using all roles in the field `clients`). The Bank expects two kinds of messages: (i) a request to create a new account (a `NewAccountReq`); and (ii) a request to share an existing account (a `Share(index)`). In both cases, a new role `r` is created using the `DefaultRoleGenerator`. In the first case, this role is simply send using a `NewAccount(r)` message; in the second case, this new role is set to both the new client and to the original owner (which forwards it to the account to share), using a `Shared(r)`. The second case fails (the Bank sends back a `Failure` message) if the owner tries to share an account with itself. Figure 5.45 shows the extract of the implementation which performs this reply mechanism. The location distinguishes two cases: first, if the message `msg` is a `NewAccountReq`, and, second, if it is a `Share(client_index)`, where `client_index` is the index of the client to share the account with. In the first case, the Bank location simply replies with a new role name (it commits a

[54] See Section 5.5.2.

[55] This field is implemented as a `Vec<Role>`, which makes it accessible by index. Also, since the set of clients is statically set, each role to a client in that set can be accessed by an integer which does not change during the program execution.

```
TransitionResult::RECEIVED(rid, msg) => {
    if let Messages::NewAccountReq = *msg {
        self.l.send(rid, Messages::NewAccount(self.rg.new_id()));
        self.l.commit_changes().pop().unwrap().unwrap();
    } else if let Messages::Share(client_index) = *msg {
        if let Some(role) = self.clients.get(client_index) {
            if *role != rid {
                let new_role = self.rg.new_id();
                self.l.send(*role, Messages::Shared(new_role));
                self.l.send(rid, Messages::Shared(new_role));
                self.l.commit_changes().pop().unwrap().unwrap();
                break 'main_loop;
            }
        }
        self.l.send(rid, Messages::Failure);
        self.l.commit_changes().pop().unwrap().unwrap();
    }
}
```

Figure 5.45: Extract of the main loop of the `BankLocation`. This extract shows the management of a message `msg` received from a role `rid`: if `msg` is a `NewAccountReq`, then the location simply commits a transition which sends a new role back to `rid`; otherwise, if `msg` is a `Share(client_index)`, and if the role at the given index is distinct from `rid`[56], the `BankLocation` instantiates a new role and sends it back to both the original and the new client, and fail otherwise. Other kinds of messages are simply ignored.

transition, built with a single item with the line `self.l.send(rid, Messages::NewAccount(self.rg.new_id()))`). In the second case, the Bank location first retrieve the role it shares with the future shared owner (the line `self.clients.get(client_index)`). If the index is not correct (the call to get does not return `Some(role)`) or if the role to the new shared owner is the role on which we received the request (i.e. if the client attempts to share the account with itself), the Bank replies with a failure (the transition built with `self.l.send(rid, Messages::Failure)`); otherwise, the Bank creates a new role name (`self.rg.new_id()`) and sends it to both the requester and the new shared owner.

Remark. The `break 'main_loop;` in the extract can be ignored. ◁

Client locations (structure `ClientLocation`) have seven fields, among which we find: (i) the underlying `Location`; (ii) a role `rid` to the Bank location; (iii) two roles, `write` and `read`, to communicate with the associated TCP locations; and (iv) a associative map `account` which maps integers to roles toward accounts (of type `HashMap<u32, Role>`). The behaviour of client location is longer to describe (but not harder) than those of the `BankLocation` and the `AccountLocations`: it can receive messages from both the TCP connection (commands from the user) and from the Bank component (in case an other client shares an account with this client). Therefore, clients continuously attempt to receive from `read` and from `rid`. Upon reception of a message, if it comes from the TCP connection[57], the (text) line is parsed and the client reacts accordingly[58]; and if the message is from the Bank[59], the client binds the new role and updates it sort accordingly.

Policy enforcement. We use exclusively the authorisation function to enforce the (variant of) owners-as-ombudsmen policy. Therefore, we do not specify the set of unconstrained location transition (we take the greatest possible set). In the actual implementation, we use the `TrivialTS` (see Section 5.5.1) transition set, and we implement a dedicated authorisation function.

Our authorisation function has to verify two points: (i) communications should only happened between authorised locations; and (ii) sort updates shall be done properly.

Communications should be allowed only between locations belonging in the same ownership domain. To verify this is the case, for each message sent (i.e. for each `TransitionItem::SEND(role, msg, dir)` in the transition), we have to verify that the recipient of the message belongs in the same ownership

[57] If the message is `Tcp(text)`.

[58] The only two non-trivial cases are when the command is (i) an account creation, in which case the client has to ask a new role name to the Bank, bind that role, and create a new account with the received role; or (ii) a request to share an account with an other client, in which case the client informs the Bank thusly, which replies with `Shared(r)` (`r` being a new role name) that the client forward to the account

[59] If it is `Shared(r)`.

```
let send_to_child_or_parent: bool = s
    .get_locations_binding(&rid)
    .iter()
    .filter(|sk_l| **sk_l != SkeletonLoc::from(l))
    .next()
    .map(|sk_l| sk_l.sort())
    .map(|sort| {
        BankAF::owns(l.sort(), &sort)
            || BankAF::owns(&sort, l.sort())
            || sort.is_tcp()
    })
    .unwrap_or(false);

if !send_to_child_or_parent && !l.sort().is_tcp() {
    return false;
}
```

Figure 5.46: Extract of the `authorise` method of the authorisation function used in the Bank system. This extract is the section of the function which searches for the recipient of a message sent on the role `rid` in the skeleton location graph `s`, and verifies, using the function `owns` (shown in Figure 5.47) that the sender `l` is the owner of the receiver, or vice-versa. It also authorises communications with TCP locations.
To find the receiver, the authorisation function browses the skeleton graph and search for the skeleton locations that bind the role on which the communication occurs. There are, at most, two such skeleton locations (the sender and the receiver). The sender is removed, by retaining skeleton locations that are different from the skeleton of the location that takes the transition. Then, using the sort of the sender and of the receiver, we check if one is owned by the other, and store the result of this verification in `send_to_child_or_parent`. If no receiver is found (the role is unbound), then we are conservative and assume that the sender sends to a role it does not own.

domain than the emitter. As a matter of fact, in our actual case, siblings never need to communicate together (no two accounts or two clients have a need to exchange messages). Therefore, we implement an even stricter policy, in which messages can only be exchanged between a parent location and one of its child locations[60]. This is done in two steps, for each message: (i) search, in the skeleton location graph, the recipient of the message (Figure 5.46); and (ii) verify that the sender is the owner of the receiver, or vice-versa[61] (Figure 5.47).

Sorts can be modified, but not all modifications are allowed. In particular, the Bank and TCP locations can not change their sort, and Client and Account locations can change their sort, under the condition that: (i) the new sort is the same variant than the old one (i.e. Account locations remain Account locations, and Client locations remain Client locations); (ii) they bind the identifier they expose, according to the owners-as-ombudsmen policy; and (iii) for Client location, the sort modification does not change the identifier. The authorisation function simulates the transition and verifies that the new sort is consistent with the old one and with the role bound. An extract of the function `check_sort`, which performs this verification, is shown in Figure 5.48.

5.6.2 An application of the logger system: a Publish-Subscribe server

In the introduction (see Section 1.2 p. 6), we presented a generic logging system where a main component needed multiple subcomponent and where each subcomponent can log its action using a logger component. In this section, we show how such system can be implemented in Rust using our library. To better illustrate this system, as well as to show the capabilities of the library, we implemented a *Publish-Subscribe Server* over TCP, which activity is logged using the presented logging system.

Publish-Subscribe Server. A Publish-Subscribe server is a server used for broadcasting messages. Messages are associated with topics. Clients can connect to the server and subscribe to some topics to receive messages broadcasted for these topics. On the other hand, clients can also publish messages for given topics, which are then forwarded to the clients that subscribe to the respective topic[62]. Figure 5.49

[60] We chose to implement this stricter policy because it is simpler, since there is no need to find a common parent in the graph when two siblings communicate. Finding a common parent using the framework is straightforward since the implementation provides functions to filter and find locations from the skeleton graph.

[61] To take into account TCP locations, that are not constrained, we also authorise messages toward/from locations with sort `TCP`.

[62] In our case, clients are allowed to send messages on topics they are not subscribed to.

```
fn owns(owner: &BankSort, owned: &BankSort) -> bool {
    match (owner, owned) {
        (BankSort::Client(_, v1), BankSort::Account(v2)) => {
            for r in v1 {
                if v2.contains(&r) {
                    return true;
                }
            }
            false
        }
        (BankSort::Bank, BankSort::Client(_, _)) => true,
        _ => false,
    }
}
```

Figure 5.47: The function `owns` which returns true if the provided `owner` sort is indeed an owner of the `owned` sort. As stated above, our ownership relation is close from the owners-as-ombudsmen one (A owns B if the sort of A contains an identifier exposed in the sort of B), except that, in addition, we consider that the Bank component is an owner of all Client locations.

```
BankSort::Client(new_id, new_owned) => {
    if let BankSort::Client(old_id, _) = old_sort {
        if new_id != old_id {
            return false;
        }
        for owned in new_owned {
            if !new_provided.contains(&owned) {
                return false;
            }
        }
        if !new_provided.contains(&new_id) {
            return false;
        }
        true
    } else {
        false
    }
}
```

Figure 5.48: Extract of the function `check_sort` which verify that a new sort (the `BankSort::Client(new_id, new_owned)` matched) is consistent with an old sort (called `old_sort`) and with the new sets of provided roles (called `new_provided`) of the location. This extract shows the case in which the new sort is the variant `BankSort::Client(_, _)`. The function successively verifies that: (i) the identifier of the client does not change; (ii) all owned identifier roles are bound in the provided direction, in the new set of provided roles; and (iii) the identifier of the client is also bound in provided direction.

```
/topic1
topic1: message 1.1
[topic1] message 1.1
[topic1] message 1.2
\topic1
```

(a) Client 1

```
/topic1
/topic2
[topic1] message 1.1
topic1: message 1.2
[topic1] message 1.2
topic2: message 2.1
[topic2] message 2.1
topic1: message 1.3
[topic1] message 1.3
\topic1
\topic2
```

(b) Client 2

Figure 5.49: A run of a Publish-Subscribe server with two clients. In this run, the command to subscribe to a topic is `/topic`; the one to unsubscribe is `\topic`; and the one to publish a message on a topic is `topic:message` (no leading space is inserted nor removed, for both the topic and the message). Messages received are printed as `[topic]message`. In this run, both clients initially register to `topic1` and Client 2 additionally register to `topic2`. Messages sent on `topic1` (`message 1.1` and `message 1.2`) are received by both peers (notice that clients receives all messages for the topics they subscribed to, even if they are the publisher). The message `message 2.1` sent on `topic2` is not received by Client 1, since it did not subscribe to that topic. After Client 1 unsubscribed from `topic1`, it does not receive messages on that channel (e.g. `message 1.3`).

shows an example of two clients exchanging messages using a Publish-Subscribe Server.

General overview. The server contains three main components: (i) a component (called *Main*) to receive interactions from clients, whether they are (un)subscriptions to some topics or message publication; (ii) a component (called *TCP Listener*), created by the Main component at initialisation, to listen for connections from new clients and which manage those new connections; and (iii) a logger component (called *Logger*), which receives logs from clients and display them. The server interacts with clients using TCP connections. The TCP Listener component listen for new connections on port 12345. Upon new connection, the TCP Listener creates three components for the new client: (i) a component (called *TCP connection*), which handles interactions with the TCP connection as explained in Section 5.5.3; (ii) a component (called *Client*), which parses commands received on the TCP connections and forward them to the Main component and which receives publications from the main component to send on the TCP connection; and (iii) a logger subcomponent (called *Logger for Client*), which is an entry point for the Client component to the Logger component. Figure 5.50 shows the overall architecture of the server.

One can clearly see that this architecture suits the logging example presented in the introduction[63]. Let alone, for the sake of simplicity, the TCP Listener and the TCP connection components, and the component graph is exactly an instance of the example presented in the introduction. Therefore, as shown in Figure 5.51, we can easily apply the method explained in Section 4.2 and the location graph for TCP connection presented in Section 5.5.3 to implement such a server in location graph. We simply have to add additional sorts and roles for the TCP related connections.

Types. Sorts of our Publish-Subscribe server are of three variants: (i) `Main` sorts, for main components, i.e. the `S` and `L` sorts presented above; (ii) `Sub` sorts, for subcomponents, i.e. the client and their logger entrypoints, which correspond to the sorts $\langle \mathtt{S}, i \rangle$ and $\langle \mathtt{L}, i \rangle$; and (iii) `TCP` sorts, used for the additionnal locations required for the TCP connections. As explained in Definition 82, we have two *sides* for sorts, `S` for the system side of the location graph, and `L` for the logger side of location graph. We therefore have an enumerated type `Side` which contains those two variants, as shown in Figure 5.52. To take into account the additionnal TCP locations, we add a type `TCPSide` to distinguish the TCP locations, the TCP readers and the TCP listener, as shown in Figure 5.53. Finally, the type of sorts is an enumerated type which uses both `Side` and `TCPSide` to implement our `Sort` type, as shown in Figure 5.54.

Roles are directly implemented from Definition 83, using `DefaultRole` presented in Section 5.5.2 as $\mathbb{R}_b$. We add two kinds of roles: `TcpToSub` and `SubToTcp` which are used between a Client and its associated TCP locations. The type of roles is shown in Figure 5.55.

The type of messages is shown in Figure 5.56. It contains a lot of variants, but it is not necessary to understand closely the meaning of each of these variants, since the policy does not inspect the messages.

[63] As a matter of fact, we identified the logging example as an interesting pattern independently from the idea of implementing a Publish-Subscribe server. The fact that the chosen architecture for the server suits the logging pattern was unexpected, which emphasises the relevance of the pattern.

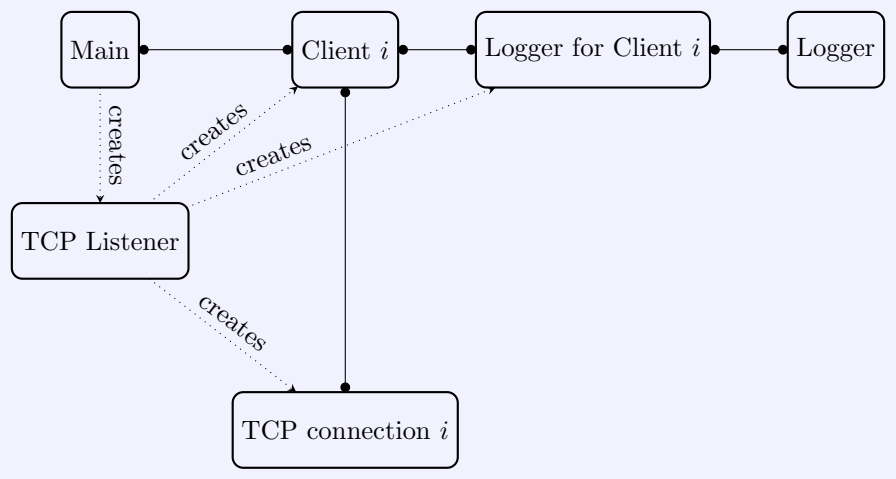


Figure 5.50: Overview of the Publish-Subscribe server. The goal of the Main component is to manage subscriptions and publications. At initialisation, it creates a TCP Listener component which waits for incoming TCP connections. When a new connection is established, it creates three subsystems: (i) a component to read and write from/to the connection (the TCP connection component); (ii) a component which both parses data received from the TCP connection and interprets it, and which listen for any broadcast from the Main component (the Client i component); and (iii) a component which logs the execution of the Client i component, and forward it to the main logger (the Logger for Client i). Finally, the Logger component aggregates and print all data received from any Logger subcomponent.

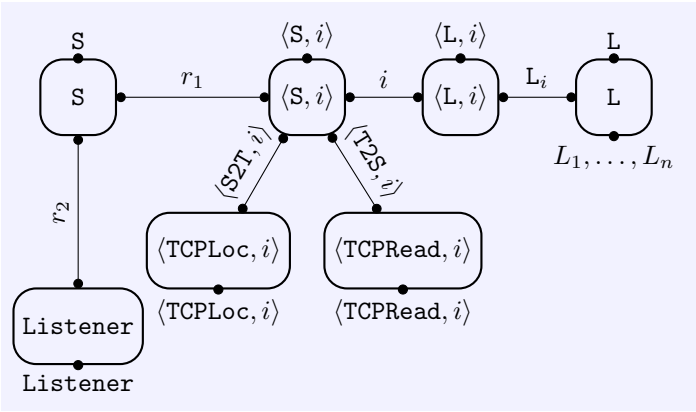


Figure 5.51: Location Graph instance for the Publish-Subscribe server. We apply directly the Location Graph explained in Figure 4.6 for the Main, Clients, Loggers for Clients and Logger components. In addition, we introduce additional sorts and roles for the TCP related components. First, the TCP connection component is implemented as explained in Section 5.5.3, which uses two locations. We introduce the additional sorts `Listener`, $\langle \mathtt{TCPLoc}, i \rangle$ and $\langle \mathtt{TCPRead}, i \rangle$ for, respectively, the TCP Listener, the TCP Location (for client i) and the TCP Reader (for client i). As usual, those location bind their sorts as provided roles to prevent duplicates. Finally, we add the role names $\langle \mathtt{S2T}, i \rangle$ and $\langle \mathtt{T2S}, i \rangle$ for the communication between the location for client i and its two TCP locations. Finally, notice that, for the sake of simplicity, the location `L` used for the main component of the logger has a fixed capacity n and binds roles $\{L_1, \ldots, \mathtt{L}_n\}$ for its whole lifetime. This choice is consistent with respect to Section 4.2.

```rust
#[derive(Ord, PartialOrd, Eq, PartialEq, Clone, Copy, Hash, Debug)]
enum Side {
    S,
    L,
}
```

Figure 5.52: The enumerated type `Side`, which implements the two elements `S` and `L` presented in Definition 82.

```
#[derive(Ord, PartialOrd, Eq, PartialEq, Clone, Copy, Hash, Debug)]
enum TCPSide {
    Loc(u32),
    Reader(u32),
    Listener,
}
```

Figure 5.53: The enumerated type `TCPSide`, which is used for the sorts of TCP related locations. The integer in the `Loc` and `Reader` variants is the identifier of the client which uses the given connection.

```
#[derive(Ord, PartialOrd, Eq, PartialEq, Clone, Copy, Hash, Debug)]
enum Sort {
    Main(Side),
    Sub(Side, u32),
    TCP(TCPSide),
}
```

Figure 5.54: The type `Sort` used in our Publish-Subscribe server.

```
#[derive(Ord, PartialOrd, Eq, PartialEq, Clone, Copy, Hash, Debug)]
enum Role {
    Base(DefaultRole),
    Int(u32),
    L(u32),
    Sort(Sort),
    TcpToSub(u32),
    SubToTcp(u32),
}
```

Figure 5.55: The type of roles used in the Publish-Subscribe server. Except for the additionnal `TcpToSub` and `SubToTcp` used for the interactions between a Client and its TCP locations, the four variants correspond to the four sets which compose $\mathbb{R}_{ls}$ in Definition 83.

```rust
#[derive(Clone, Eq, PartialEq, Ord, PartialOrd, Debug)]
enum Message {
    S(String),
    Administrative(String),
    Shutdown,
    Pub(String, String),
    NewSub(Role),
    Sub(String),
    Unsub(String),
}
```

Figure 5.56: The type of messages used in the Publish-Subscribe server. The variants are used as follow: (i) `S(String)`, which is either an arbitrary message emitted by a client to be logged or a message received over the TCP connection, sent by the TCP location to the Client; (ii) `Administrative(String)`, which is an administrative message emitted by a logger subcomponent to be logged; (iii) `Shutdown`, which is propagated among locations related to a particular Client when the connection is closed; (iv) `Pub(String, String)`, which is exchanged between Clients and the Main component to indicate a message is published on a given topic; (v) `NewSub(Role)`, which is sent from the TCP listener to the Main component when a new Client is created, in order to provide the Main component a role toward this new Client; (vi) `Sub(String)`, which is sent from a Client to the Main to indicate that the Client subscribes to the given topic; and (vii) `Unsub(String)`, which, on the contrary, indicates that a Client unsubscribes from a given topic.

Locations. Locations are split among four structures, with the addition of the three TCP locations presented in Section 5.5.3. The four structures are, in the order we will present them:

1. `SubComp`, which implements a Client component
2. `Main`, the main system component
3. `LoggerComp`, which implements the logger entrypoint of a Client component
4. `Logger`, the main logger component

The `SubComp` (see Figure 5.57) location continuously attempt to receive messages from either the `Main` location or the associated `TcpReader` location. Upon reception of a message, (i) if the message comes from the associated `TcpReader` location, if that message is a `S(String)`, the message is parsed into the corresponding command message (a `Pub(String, String)`, a `Sub(String)`, or a `Unsub(String)`) and is forwarded to the `Main` location, or if it is a `Shutdown`, the loop breaks and the client terminates; or (ii) if the message is from the `Main` location, and that message is a `Pub(topic, message)`, then a corresponding `String` is formatted and sent to the corresponding `TcpLoc`. In any case, the action is logged: an adequate message is sent to corresponding `LoggerComp` location. The `SubComp` location has a method `local_log` used to send a message to the corresponding logger (see Figure 5.58).

The goal of the `Main` location is to keep track of client subscriptions and to broadcast messages accordingly. The `Main` structure (shown in Figure 5.59) contains a field `roles` which has type `HashMap<Role, Vec<String>>`. This structure holds, for each role from the `Main` location to a `SubComp` location, the set of topic the `SubComp` has registered too. In addition, it has a separate role toward the `TcpListener` (the field `to_listener` in order to receive updates when a connection is establish to a new client. In its main loop, this location (see Figure 5.60) listens for messages from all known client (i.e. from all roles in the `HashMap<Role, Vec<String>>` maintained) and from the listener. Upon reception of a message, the location updates its hashtable, in the case of a `Sub`, `Unsub`, `NewSub`, or `Shutdown` message; or publish it, in case of a `Pub` message. To publish a message, it simply sends it to all roles in the hashtable that have the relevant topic in the associated set of topics.

Each `SubComp` is associated to a `LoggerComp`. The `LoggerComp` (see Figure 5.61) continuously listens on the role `Int(i)`, where `i` is the integer in its sort. Upon reception of a message from the associated `SubComp`, the message is forwarded to the `Logger`.

The `Logger` structure (see Figure 5.62) has an initial `capacity` used to indicate the number of roles the logger listens to. The `Logger` initially binds all roles `L(i)` with `i` smaller than the capacity and then continuously waits to receive logs from any of those roles. Messages received are either `S(msg)` (for regular logs) or `Administrative(msg)` (for logging `LoggerComp` activity itself). Upon reception of any of those two messages, the `Logger` formats it accordingly and displays it on `stderr`. Figure 5.63 shows the main loop of the logger.

Remark. When we described the behaviour of the `TcpListenerLocation` location in Section 5.5.3, we saw that it is implemented as a *location with holes*, where the real work is left empty so that the user can

```
for trans_result in result.unwrap() {
    match trans_result {
        TransitionResult::CREATED(_) => {}
        TransitionResult::RECEIVED(rid, m) => {
            if rid == self.tcp_to_sub {
                match *m {
                    Message::Shutdown => {
                        self.local_log("Shutdown.".to_string());
                        break 'main;
                    }
                    Message::S(topic) => {
                        self.local_log(format!(
                            "Received␣\"{}\"␣from␣TCP.",
                            &topic
                        ));
                        let msg = string_to_msg(topic);
                        if let Some(m) = msg {
                            self.l.send(self.sub_to_main, m);
                            self.l.commit_changes();
                        }
                    }
                    _ => {}
                }
            } else {
                // Received from sub_to_main
                match *m {
                    Message::Pub(topic, message) => {
                        self.local_log(format!("[{}]{}", topic, message));
                        self.l.send(
                            self.sub_to_tcp,
                            Message::S(format!("[{}]{}\n", topic, message)),
                        );
                        self.l.commit_changes();
                    }
                    _ => {}
                }
            }
        }
    }
}
```

Figure 5.57: The part of the main loop of the `SubComp` location structure which evaluates the result of the transitions attempted. No location creation is attempted, and therefore no `CREATED` transition result can be received: thus we simply ignore any of those. In case a `RECEIVED(rid, m)` is in the result of a transition, we look at `rid` which indicates which location sent the message: if the message comes from the `TcpReader`, it can takes two forms: either a `Shutdown`, in which case we break the loop; or a `S(msg)`, in which case we parse (using the method `string_to_msg`, which we do not show here) the string `msg` to obtain a command to send to the `Main`; otherwise, `rid` is the role toward de `Main` location, and the message can only be a `Pub(topic, message)`, which is then formatted and sent to the `TcpLoc` location.

```rust
fn local_log(&mut self, s: String) {
    if let Sort::Sub(Side::S, i) = *self.l.sort() {
        self.l.send(Role::Int(i), Message::S(s));
        self.l
            .commit_changes()
            .pop()
            .expect("Could not find the result of the transition.")
            .expect("Could not find the result of the transition.");
    }
}
```

Figure 5.58: The `local_log` method of the `SubComp` structure. This method simply embeds a given string in a `S` message and sends it (blindly) to the associated logger location, using the role `Int(i)`. Notice the two `expect` lines: the method `commit_changes` returns a vector of results, from which we take the first value (method `pop()`); the method `pop()` possibly fails if the vector is empty (which never happens in our case): it therefore returns an `Option<T>`, where `T` is the type of values in the vector, in our case `Result<_, _>`. The first `expect` opens the `Option`, the second the `Result<_, _>`.

```rust
struct Main {
    l: Location<Sort, LoggerAF, Message, Role, LoggerTS>,
    roles: HashMap<Role, Vec<String>>,
    to_listener: Role,
    rg: Option<DefaultRoleGenerator>, // Kept only between instantiation and
    ↪ initialisation.
}
```

Figure 5.59: The `Main` structure, used to implement the Main location of our system. The field `roles` associates each known role toward a `SubComp` the set of topic the corresponding client subscribed to. The field `to_listener` is used to receive, from the `TcpListenerLocation`, the roles to the new client upon connection.

specify it. In this example, and for the sake of conciseness[64], we do not show this part in detail: one just need to know that, upon new connection, the `TcpListenerLocation` creates the four locations for the new client: the `SubComp`, the `LoggerComp`, and the two TCP locations `TcpReader` and `TcpLocation`. ◁

> As it is right know, there is no guarantee of forward progress. Actually, there is a possibility of deadlock if: (i) a `SubComp` sends a command to the `Main`; and (ii) the `Main` broadcasts a message to multiple client, including the one above. This is due to the fact that, when broadcasting a message, the `Main` component proposes a transition in which it tries to send the message to all relevant clients. Also, a when a `SubComp` sends a command to the `Main` component, it proposes a transition which sends a (command) message to the `Main` component. Therefore, if both events happen simultaneously, the system ends up in a state in which both the `Main` and the `SubComp` components propose a transition which is not matched. Therefore, the system deadlocks.
> For the sake of simplicity, we ignore this possibility, since we are more concerned about safety than liveness. Also, fixing such problem would not be technically hard, but is not relevant for the purpose of demonstrating the safety of the policy.

Policy enforcement. To enforce the logging policy, we take the same approach as presented previously (see Section 4.2): with a carefully specified set of unconstrained location transitions, it is possible not to use the authorisation function. Of course, we have to adapt this set of unconstrained location transitions to take into account the additional locations required for the managment of TCP connection. All in all, specifying this set (i.e. writing a structure and implementing the `TransitionSet` trait) took around one or two hour(s)[65] and the result take around 200 lines of code (the code was not optimised for conciseness, but for clarity instead, which yields longer code).

Remark. Our adaptations to take into account TCP related locations are the following: (i) locations with sort $\langle \texttt{TCPLoc}, i \rangle$ are allowed to bind $\langle \texttt{S2T}, i \rangle$, in addition to their sort; (ii) locations with sort $\langle \texttt{TCPRead}, i \rangle$

[64] And also because there is no hidden black magic.

[65] Excluding the time to write the `binds_only` and `forbid_self_removal` utility functions, which are now available in the `utils` module.

```
for r in res {
    match r {
        TransitionResult::RECEIVED(rid, m) => match *m {
            Message::Pub(topic, message) => {
                self.submit(topic, message);
            }
            Message::Sub(topic) => {
                self.roles.get_mut(&rid).unwrap().push(topic);
            }
            Message::Unsub(topic) => {
                self.roles.get_mut(&rid).unwrap().retain(|t| *t != topic);
            }
            Message::NewSub(r) => {
                self.l.bind(r, RoleDirection::PROVIDED);
                match self.l.commit_changes().pop().unwrap() {
                    Ok(_) => {
                        self.roles.insert(r, Vec::new());
                    }
                    _ => {}
                }
            }
            Message::Shutdown => {
                self.roles.retain(|r, _| *r != rid);
            }
            _ => {}
        },
        _ => {}
    }
}
```

Figure 5.60: Extract of the body of the loop of `Main`. This location attempts to receive commands from all known roles (not shown) and, when a transition is taken, the extract presented here is executed. Depending on the kind of the command, the `HashMap` `self.roles` is updated (in the case of `Sub`, `Unsub`, `NewSub` and `Shutdown`); or, in the case of `Pub`, the message is broadcasted using the method `submit` (not shown).

```
self.l.receive(Role::Int(i));
match self
    .l
    .commit_changes()
    .pop()
    .expect("Could␣not␣find␣the␣result␣of␣the␣transition.")
{
    Ok(msgs) => {
        for msg in msgs {
            match msg {
                TransitionResult::RECEIVED(_, m) => match *m {
                    Message::Shutdown => {
                        self.local_log("Shutdown.".to_string());
                        return;
                    }
                    _ => {
                        self.l.send(Role::L(i), *m);
                        self.l.commit_changes();
                    }
                },
                TransitionResult::CREATED(_) => {
                    self.local_log(String::from(format!(
                    "Received␣an␣unexpected␣location␣creation␣transition␣result"
                )))
                }
            }
        }
    }
    Err(e) => self.local_log(String::from(format!(
        "Trying␣to␣receive␣from␣{:?}␣failed:␣{:?}",
        Role::Int(i),
        e
    ))),
}
```

Figure 5.61: The body of the loop of the `LoggerComp` location. This location continuously waits for a message on role `Int(i)` and, upon reception performs one of the following: (i) if the message is `Shutdown`, then the location quits; and (ii) otherwise, the message is simply forwarded to the `Logger` location.

```
struct Logger {
    l: Location<Sort, LoggerAF, Message, Role, LoggerTS>,
    capacity: u32,
}
```

Figure 5.62: The `Logger` structure which includes a field `capacity` to represent the maximum number of logger subcomponents it can manage.

```
for i in 0..self.capacity {
    self.l.new_transition();
    self.l.receive(Role::L(i));
}

let mut i = 0;
for transition_result in self.l.commit_changes() {
    match transition_result {
        Ok(msgs) => {
            for msg in msgs {
                match msg {
                    TransitionResult::RECEIVED(_, m) => match *m {
                        Message::S(s) => self.log(format!("[Component␣{}]:␣{}", i,
    ↪ s)),
                        Message::Administrative(s) => {
                            self.log(format!("[Logger␣{}]:␣{}", i, s))
                        }
                        _ => self.log(format!(
                            "[Logger]:␣Received␣unexpected␣{:?}␣from␣{}",
                            m, i
                        )),
                    },
                    TransitionResult::CREATED(_) => {
                        self.log("[Logger]:␣Received␣unexpected␣CREATED".to_string
    ↪ ())
                    }
                }
            }
        }
        Err(TransitionError::NotSelected) => {}
        Err(e) => {
            self.log(format!("Can␣not␣receive␣from␣{}:␣{:?}", i, e));
        }
    }
    i += 1;
}
```

Figure 5.63: The main loop of the `Logger` location. The first step consists in proposing the reception of a message for any role `L(i)`, each time on a different transition. Only one of those transitions can be selected at a time. Once a transition is taken, the result is deconstructed to access the content of the message: an `Administrative` message is generated by the `LoggerComp`, while a `S` message is generated by the `SubComp` and the `LoggerComp` is just a proxy. Depending on the message, the `Logger` formats the string accordingly and displays it on `stderr` (in the function `log`, not shown).

```rust
fn contains(
    &self,
    t: &Transition<Sort, Message, Role>,
    l: &Location<Sort, LoggerAF, Message, Role, LoggerTS>,
) -> bool {
    if !LoggerTS::common_constraint(t, l) {
        return false;
    }

    match *l.sort() {
        Sort::Main(Side::L) => {
            return LoggerTS::l_constraint(t, l);
        }
        Sort::Main(Side::S) => {
            return LoggerTS::s_constraint(t, l);
        }
        Sort::Sub(Side::L, i) => {
            return LoggerTS::sub_l_constraint(t, l, i);
        }
        Sort::Sub(Side::S, i) => {
            return LoggerTS::sub_s_constraint(t, l, i);
        }
        Sort::TCP(TCPSide::Loc(i)) => {
            return LoggerTS::tcp_loc_constraint(t, l, i);
        }
        Sort::TCP(TCPSide::Reader(i)) => {
            return LoggerTS::tcp_read_constraint(t, l, i);
        }
        Sort::TCP(TCPSide::Listener) => {
            return LoggerTS::listener_constraint(t, l);
        }
    }
}
```

Figure 5.64: The `contains` method of the `LoggerTS` structure. This method first calls `common_constraint`, and if no violation of the common constraints are found, it then proceeds to the verification of constraints based on the sort of the location taking the transition.

are allowed to bind $\langle \texttt{T2S}, i \rangle$, in addition to their sort; and (iii) the location with sort `Listener` is allowed to bind roles in $\mathbb{R}_b$, in addition to its sort. ◁

Our structure (called `LoggerTS`) is trivial, as it does not need any field. Concerning the `contains` method (required by the `TransitionSet` trait), there are two kinds of requirements for the set of unconstrained location transitions: those which depends on the location (e.g. which role bindings are allowed for a given location) and those which apply to all locations (e.g. all locations have to binds their sort to ensure sort uniqueness); therefore the `contains` method is split in two parts, as shown in Figure 5.64: (i) a call to a function `common_constraint`, independently of the location; and then (ii) a call to a specific function, depending on the sort of the location (e.g. `sub_s_constraint` for the location with sorts `Sub(S, _)`).

The function `common_constraint`, shown in Figure 5.65, simply loops over all transition items of the transition, and verifies that it is consistent with the constraints we want to enforce (eg, if a `TransitionItem::Sort(s)` is found, with an `s` different from the current sort of the location taking the transition, then it forbids the transition; this particular example ensures that locations can not change their sort).

Finally, concerning specific functions we have to check that they bind only the roles they are allowed (e.g. , locations with sort `Sub(L, i)` are only allowed to bind `Int(i)`, `Sort(Sub(L, i))`, and `L(i)`) and, for the Main component and the Logger, that they do not terminate. All specific functions look alike, and an example is shown in Figure 5.66.

```
TransitionItem::SORT(s) => {
    if **s != *l.sort() {
        return false;
    }
}
TransitionItem::CREATE(s) => {
    // Be sure that each new <S, i> is matched by a <L, i> and vice-versa
    let new_sorts = t.into_iter().filter_map(|t_item| {
        if let TransitionItem::CREATE(s2) = t_item {
            Some(s2)
        } else {
            None
        }
    });

    if let Sort::Sub(Side::S, i) = **s {
        if new_sorts
            .filter(|s2| ***s2 == Sort::Sub(Side::L, i))
            .next()
            .is_none()
        {
            return false;
        }
    } else {
        if let Sort::Sub(Side::L, i) = **s {
            if new_sorts
                .filter(|s2| ***s2 == Sort::Sub(Side::S, i))
                .next()
                .is_none()
            {
                return false;
            }
        }
    }
}
TransitionItem::RELEASE(Role::Sort(_), _) => {
    return false;
}
TransitionItem::BIND(Role::Sort(s), RoleDirection::PROVIDED) => {
    if s != l.sort() {
        return false;
    } else {
        // If l binds its sort, remember it
        bind_sort = true;
    }
}

_ => {}
```

Figure 5.65: Extract of the `common_constraint` function. This extract is the body of the pattern matching over transition items of the transition. In turn, we can see that: (i) sorts can not be modified; (ii) creation of a location with sort `Sub(S, i)` must be matched by the creation of `Sub(L, i)`, and vice-versa; (iii) locations can not release the role that contains their sort; and (iv) locations can bind role with a sort, as long as it is the sort they carry.
Outside this loop (not shown) we additionally check that locations do bind their sort, if not bound already. This is required because, contrary to the theoretical framework, new locations in our implementation can not be created with roles already bound; we therefore have to check that each location binds their sort during their initial transition.

```rust
fn l_constraint(
    t: &Transition<Sort, Message, Role>,
    l: &Location<Sort, LoggerAF, Message, Role, LoggerTS>,
) -> bool {
    forbid_self_removal(t, l)
        && binds_only(t, |r, d| match r {
            Role::Sort(Sort::Main(Side::L)) if *d == RoleDirection::PROVIDED =>
    ↪ true,
            Role::L(_) => true,
            _ => false,
        })
}
```

Figure 5.66: The function `l_constraint` which checks the specific constraints for locations with role `Main(L)`. This function uses the functions `forbid_self_removal` and `binds_only` of the `utils` module to prevent the removal of the location and the binding of unwanted roles. In this case, this location is allowed to bind its sort (to preserve uniqueness) and roles toward logger subcomponent, i.e. those with name `L(_)`.

Chapter 6

Conclusion

6.1 General conclusion

In this thesis, we studied two aspects of location graphs.

First we studied various notions of encapsulation for location graphs. We showed that the framework can accommodate a wide range of different forms of encapsulation. To emphasize that those notions of encapsulation indeed relate to a form of information hiding, we developed a generic method to nest some parts of the graph into bigger aggregates. Our main result of this first part is that the nested version of the graph behave correctly with respect to the original version. In addition, we showed that, even with notions of encapsulation that do not have a unique intuitive partitioning of objects (typically in the case of sharing), our method is such that the multiple possible views of the encapsulation can co-exists, and that we can easily switch from one to another, even during the run.

To perform this analysis, we had to introduce a new notion of (bi)simulation, to relate instances from different models of location graphs.

In the second part of this thesis, we implemented the location graph framework as a Rust library. We first defined an abstract machine for location graphs, to detail the primitives available to the user, as well as their behaviour. This abstract machine is then implemented as a Rust library, and allows any user to program with the location graph paradigm, such as one can program with the actor model paradigm using e.g. Akka in Scala.

We illustrate the usage of this library with various examples, two of which being presented in this manuscript.

On the notion of encapsulation. To achieve the first part of our work, which is to characterize the notion of encapsulation, we had to proceed in two steps. We wanted to characterize encapsulation with a behavioural equivalence, therefore our first step was to formalise the notion of behavioural equivalence; our second step was to characterise the notion of encapsulation; and finally, we had to prove that this notion of encapsulation was indeed correct, with respect to the behavioural equivalence. To our knowledge, the characterisation of encapsulation as a behavioural equivalence is original. Traditionnally, encapsulation is characterised with a notion of information hiding/absence of information leaks. I think a behavioural approach is more flexible, in that it does not assume anything on the component graph.

Our notion of behavioural equivalence is characterised as a *simulation relation*. More precisely, we introduced the notion of *heterogeneous simulation* and we adapted the notion of *partial bisimulation*, to end up with a notion of partial heterogeneous bisimulation. Heterogeneous simulation allows us to compare location graphs with different basic types, thanks to equivalence relations between environments and labels each instance. On the other hand, partial bisimulation is a weaker form of bisimulation, in which some labels can be ignored. Finally, a partial heterogeneous bisimulation is a relation which is an heterogeneous simulation and which inverse is also an heterogeneous simulation, up to some labels.

We defined encapsulation with a notion of nesting: we split a location graph into multiple subgraphs, and we nest each subgraph in individual locations. This forms a new location graph, which processes are graphs. We gave a formal description of this process. In addition, we had to specify the semantics of such nested location graphs, which are expressed as location graph semantics; which emphasize that nested location graphs *are* location graphs.

Finally, we showed that nested location graphs are bisimilar to their flat counterpart, using our notion of bisimulation.

The proof is not trivial, but we can summarise it intuitively. The most important step is to show that the way we split the graph does not matter, and that when reducing, we can exchange location at will. Proving the bisimulation result in itself is quite classic: we show separately that the nested graph

simulates the flat graph (which is easy, since we constructed the nested graph with that in mind), and that the flat graph simulates the nested graph, if no administrative labels are found. The difficult part is that, even if no administrative labels are found at the base level, we can not deduce that no such label appears at all: in particular, if two matching labels are present, they are erased. Therefore, we can not perform a standard proof by structural induction. Fortunately, even if such administrative labels appears internally, we show that we can remove them, until no such label are found (said otherwise, we can always find an other way to split the graph which generates no administrative labels, at any level). With that, concluding the proof is quite mechanical.

With our new notion of encapsulation, we studied how classic isolation policies would apply.

First we studied the family of ownership-based type systems, used to implement hierarchical structures (with sharing). In this first set of example, we implemented location graphs which guarantee the same isolation invariants than the ownership type systems. We showed that this is indeed a form of encapsulation. In addition, we showed that those invariant are indeed close to each other, and slight variation on the ownership relation are sufficient to switch from one to an other. In those first examples, we ensured our invariants thanks to the authorisation function.

Second, we implemented an ad-hoc policy, in the form of a simple logging service. This policy is not hierarchical, and there are multiple ways to encapsulate its components. We showed that the location graph framework is expressive enough to accommodate such non-hierarchical structures. In this second example, we enforced the invariant with a careful specification of the location reduction rules, contrary to the first example. Therefore, for that example, there is no need for the authorisation function. This illustrate the trade-off one designer faces: putting the complexity in the semantics or in the dynamic verification.

On the implementation. The second part of our work is the implementation of the location graph framework in Rust, and the implementation of some of the examples we presented in this thesis.

Rust was chosen for convenience. The isolation guarantees provided by its type system proved useful; in particular, a first attempt of implementation in Java, which was much more limited than the current one, was much longer, in terms of lines of code, and of higher complexity. The Rust programming language allows the current implementation to be less than 3000 lines of code long (including utilities, including inline documentation, excluding examples), which is quite small.

The library is composed of two main part, following the location graph spirit: an authorisation function part, and a location part. The authorisation function simply is a trait (an interface), than one has to implement, which decides whether a transition is authorised or not. The location part, on the other hand, provides a few primitives to allow a location to take transitions. In our library, a location transition is an aggregate of transition items, such as message exchanges or graph modification. When a transition is fired, all or none of the items are performed, in a transaction style. In addition, a single location can propose multiple alternative transitions at once, only one of those being chosen by the system.

Internally, the operation is quite simple, yet computationally expensive: since locations can propose multiple alternative transitions, we compute all possible combinations to find a suitable one (i.e. a complete combination, correct with respect to the chosen authorisation function). In addition, we maintain a skeleton of the location graph, which is used to evaluate the authorisation function.

Even if our implementation is not optimised, I want to emphasize that it is not just a demonstration example nor a research artifact. In particular, it assumes almost nothing about the basic types (for instance the types of messages can be anything, and not just integer or such), and the constraints are forced by technical reasons: typically, when two locations synchronise on a value, the type of that value must provide a notion of equality.

Finally, we produced two examples which illustrate two isolation policies that we followed through this manuscript. The first example is an example of a bank system, which illustrate a form of shared ownership. The second example is a Publish-Subscribe server, which illustrate an ad-hoc isolation policy. Those two examples are quite short (respectively 680 and 830 lines, all included). Implementing those two examples showed that, even for distinct examples, some construct would be common, such as locations to establish TCP connections, which are to be used in servers. We grouped those common constructs in a utility module, so that it can be used by others.

6.2 Future work

6.2.1 Short term

Along this manuscript, we left opened some problems as future work. In this section, we review all those problems, which I expect to be easily solvable (for instance, during an internship or a master thesis).

Higher-order nesting. As a conclusion of the section on nesting 3.3.4, we informally presented an intuition for higher-order nested graphs. An important step forward would be to extend our partial bisimulation result (Theorem 4) to such nesting. Of all the presented short term goals, this one is probably the most uncertain (and also the more interesting).

Improving the confidence in our results. Mechanisms and results on location graphs presented in Chapter 3 are quite intuitive, but their formalisation, which introduce tens of definitions and lemmas, is quite hard to follow and is prone to error. While I am quite confident in the overall correctness of the theorem shown, the complexity of the framework makes me almost sure that some errors (or some unintended shortcuts) are present in the proofs, despite all the care put in proofreading the manuscript.

As a consequence, formalising the theoretical work presented would greatly increase the confidence we can have in the result presented. While proving the elements of this work, I always kept in mind this future step, detailing every step so that the proof could (hopefully) be easily ported into a proof assistant such as Coq.

Concerning formal verification, the other side of the coin is proving that our implementation is actually correct. There are two steps to do achieve this goal: proving a correspondence between the formal model of location graphs and our abstract machine, and then proving that our implementation indeed implements the abstract machine.

The main problem of this is that, as of today, the infrastructure to verify Rust code is at its very early stages (remember that Rust was first released in 2015). Promising tools include, for instance, Prusti [6], however the results are quite recent (the initial commit of the Prusti project dates from December 2017, their presentation paper dates from October 2019).

That being said, the correspondence between the framework and the abstract machine can already be conducted using more traditional, such as Coq.

Improving the usability of our implementation. I do not claim that our implementation is the fastest, nor the smartest. I only claim it is the first one, and as such, I intend it to be more a prototype than a production ready product. While I will discuss in the following section the long term vision I have for that implementation, I can present here some possible improvements that are directly accessible.

First, a useful tool would be the ability to merge different `LocationSystem`s. This can be achieved in different ways. For instance, we could develop some ad-hoc locations (like the ad-hoc TCP locations we presented) so that different instances could communicate together. An other approach could be to actually reuse our theoretical work on nesting, and to explore how could such a system be implemented[1].

Second, with the development of more examples, one could try to figure out some common patterns and integrate them in our utility library. Location graphs present a novel way to program, and I am very curious of which patterns will emerge. Similarly to TCP locations, which addition to the utility library was driven by the needs we had to implement our examples, I expect that more generic locations, authorisation functions, transition sets, etc. can be added in a short term future.

Finally, we anticipated the usage of matching errors in Section 5.2.2, but we did not implement a detection mechanism. The implementation of that mechanism will surely prove useful in practice.

6.2.2 Perspectives

To conclude this thesis, let us have a word about long term perspectives of this work.

Encapsulation as a locality property. Considering encapsulation policies and isolation, this work exhibits a direct relation between the two notions, by relating a policy, implemented with an authorisation function and a notion of nesting.

Here, we can have a philosophical debate about whether all forms of nesting should be understood as *encapsulation*. With our method, we can create a similar graph with any partitioning of the graph, whether or not this partitioning makes sense. Think, for instance, of a chaotic location graph instance: even with such instance, without apparent notion of encapsulation, we can group arbitrary locations together and find a similar nested graph. Of course, there is no hierarchy, and such nesting would not be very useful.

However, when we talk about encapsulation (even in relaxed policies), we usually have more than just grouping arbitrary objects together. I think that the additional notion we informally think of is a notion of *locality*: we have the mental image that actions of an object are only constrained by objects in the same group and with objects it interacts with. Of course, the very introduction of this thesis intended to show that there is not a single good notion of encapsulation.

[1] We actually showed that this can be achieved naively: our problem here should focus on practical concerns, a naive implementation would surely be unusable in practice.

I think a good way to pursue this work would be to search for some classes of authorisation function that would correspond to usual notion of encapsulation. Typically, we would be interested in a class of authorisation functions which depends on a subset of the skeleton graph. This would characterise locations where we can abstract away some knowledge of the graph, and this would give us an obvious partitioning function.

The practical implication of local authorisation functions. In addition to its theoretical utility, such a class of authorisation function would also be useful in practice: if we remember our implementation, we have to lock the whole skeleton graph each time a transition is proposed, so that the evaluation of the authorisation function is atomic with the evaluation of the transition. With a *local* authorisation function, all of a sudden, we are not forced to lock the whole skeleton graph: we can simply lock the area of the skeleton graph the authorisation function depends on.

This is of prime importance, since this would mean that we can find a *local* partitioning function: that is we nest together locations that depend on each other. With such property, I would not be surprised if we could implement a lock-free evaluation of the authorisation function.

Also, an other view of such approach is that, if we aim at distributing the location graph, we have a very natural view of which locations should be on the same node.

Of course, all this is just an extension of the nesting method developed in this thesis, but specialising on a function would allow us to do more assumptions about the instance, and therefore, a dedicated implementation would be more efficient.

A distributed implementation. So far, the implementation we presented in Chapter 5 is only a library for Rust. While behind motivated by encapsulation problems, typically in object oriented languages, I think the framework is also suitable to think about distributed systems.

Therefore, it would be an interesting direction to implement a distributed virtual machine for the framework. For instance, such virtual machine could be based on Beam, the Erlang virtual machine [24]. Of course, such endeavour requires an effort to provide a distributed algorithm for the verification of the authorisation function, which relates to the points above.

As we showed in Chapter 4, some policies can be implemented using either the authorisation function or the set of unconstrained location transition. Since the unconstrained location transition does static verification, in the sense that the verification does not need an view on the current state of the graph (i.e. the set of unconstrained location transition is provided, and we just have to verify if the transition is in that set), it would be very interesting to have an automated way to take some of the verifications performed by the authorisation off, and to perform them using the set on unconstrained location transition. Such verifications could require modifications on the graph (e.g. the addition of new roles to exchange new administrative messages related to the verification). More formally, given an instance $\mathbb{G}$, the goal would be to find a new instance $\mathbb{G}'$, such that both models are bisimilar and such that the authorisation function of $\mathbb{G}'$ puts less constraints.

A word on bisimulation. In the introduction of the location graph framework, we showed that the notion of bisimulation we had was not suitable to compare heterogeneous instances of location graphs, and we therefore introduced a new notion of *external bisimulation*.

While I have a good intuition of what behaviour I wanted for this bisimulation, and I am quite confident this new notion is useful, we lack some ground theory of the bisimulation itself, even in the context of other languages. Said otherwise, the definition we have right now is very ad-hoc, but I think it could be generalised and useful in other languages. We also mentioned a notion of *internal (bi)simulation* we started (see Appendix B), which is not satisfying right now, even though it takes an interesting approach.

The domain of (bi)simulations is quite dense, with a lot of subtle variants, and I definitively do not have an understanding of the field deep enough to be able to confidently relate our new notion with existing ones, even tho this question is of prime importance. To be honest, I have the feeling that a major part of this thesis (if not the whole thesis) could have been on the study of our new bisimulation and its relation with other kinds of bisimulations and its applications on other languages.

Appendix A

Rust in a nutshell

At first sight, Rust programs may be hard to read. With a few notes on what is important and what can be safely ignored, a non-Rust programmer should be able to get the essence of programs. The goal of this appendix is to give such programmer the minimal set of tools to understand the examples given in the thesis.

A.1 Quick introduction to Rust

We first introduce the main concepts of Rust.

Types. In Rust, for each type `T`, there are four variants:
- `T` an immutable value of type `T`;
- `mut T` a mutable value of type `T`;
- `&'a T` a pointer to an immutable value of type `T`, defined in the scope (a.k.a. lifetime) `'a`;
- `&mut 'a T`, idem, except that the value is mutable.

For references, there might not be, at the same time, multiple mutable references, or a mutable reference and immutable references.

Remark. Some obvious lifetimes can be inferred by the compiler, in which case they can be elided. ◁

Furthermore, when using generic types, it is possible to specify some trait constraints – see below – on the generic, e.g. writing e.g. `T: Copy + Send`, meaning that we accept any type `T` as long as it implement `Copy` and `Send`.

Structures and objects. As usual, Rust allows to write structures, which are aggregates of values. The object aspect of structures comes with implementation blocks, which allows to write methods and associated functions (see Figure A.1).

Traits. Rust traits are mostly like interfaces in Java: their goal is to define a common behaviour to multiple structures, via collecting methods together.

Among the differences with Java interfaces, some differences are:
- Anyone can implement a new trait for a structure.
- Trait can include constants or associated types.
- The return type of methods may depends on the implementation.

For instance, in Figure A.2, we define a trait `MyTrait`. One of the differences with Java interfaces is that structures and traits are less tightly coupled: the writer of a structure can implement a trait written by someone else (as in Java), and the writer of a trait can implement its trait for structures written by someone else (See Figure A.3).

Associated types are quite complementary to generic types. One can read more informations about the similitudes and differences in the following documents:
- The first paragraph of `https://doc.rust-lang.org/book/ch19-03-advanced-traits.html`
- The answer, which gives an intuition of when to use generics and associated types: `https://stackoverflow.com/a/32065644/6022274`

Copying with `Copy` and `Clone`. The two standard traits `Copy` and `Clone` indicate two ways of copying memory: to indicate that a structure can be copied by copying bits (like using memcpy), the trait `Copy` is used; when copy requires a special special procedure (e.g. incrementing the counter of a reference counting pointer), the trait `Clone` is used. The compiler implicitly calls `Copy`, while cloning is explicit.

Remark. `Copy` implies `Clone`. ◁

```rust
struct MyStruct<T> {
        x: T,
        y: Vec<T>,
}

impl<'a, T> MyStruct<T> {
    fn push(&mut self, mut t: T) {
        std::mem::swap(&mut t, &mut self.x);
        self.y.push(t);
    }
    fn head(&'a self) -> &'a T {
        &self.x
    }
}
```

Figure A.1: Example of a structure `MyStruct` and two methods `push` and `head`. Notice that the structure is generic over type `T`. Also, notice the usage of lifetimes (in this case, lifetime could be elided, as the compiler would be able to infer them).

```rust
trait MyTrait: SubTrait {
  type AssociatedType;
  const CONST: &'static str = "Hello, World";

  fn trait_method(self) -> Self::AssociatedType;

  fn default_implementation(&self) -> () {
    println!("Default implementation.");
  }
}
```

Figure A.2: Definition of a Rust trait. This trait `MyTrait` depends on an other trait `SubTrait`: all implementors of `MyTrait` must implement `SubTrait`. The definition includes an associated type `AssociatedType`, and two methods, among which one has a default implementation. Notice that the return type of `trait_method` depends on the associated type, and therefore on the implementation.

```rust
impl SubTrait for std::vec::Vec<T> {}

impl<T> MyTrait for std::vec::Vec<T> {
  type AssociatedType = T;

  fn trait_method(mut self) -> Self::AssociatedType {
    // ... returns a T
  }
}
```

Figure A.3: Writers of traits can implement them for already existing structures, allowing them to extends their capabilities. Notice that one does not need to implement `default_implementation`.

```
let a: MyType = MyType {...};
f(a); // the compiler expands to f(a.copy())
f(a); // the compiler expands to f(a.copy())
```

Figure A.4: An example of implicit call to `copy` inferred by the compiler. Notice that this happens not only in function calls, but also in aliases, etc..

Therefore, for a new programmer, the difficulty is not to understand the difference between copying and cloning, but to be able, at places where nothing is written, to know whether an implicit copy is performed, or if a transfer of ownership is done.

Consider, for instance, the example in Figure A.4. The programmer instantiate an object `a` which has type `MyType`, and then calls the function `f` twice, giving each time the object `a`. If `MyType` *does not* implement `Copy`, then the first call `f(a)` take ownership of `a`, and the second call causes an error, since the current block *does not* own `a` anymore. Therefore, such example requires that `MyType` implements `Copy`, and the object `a` is implicitly copied. An other possibility is to implement `Clone` and to explicitly call `f(a.clone)` instead of the first `f(a)`.

Destructors with `Drop`. It is possible to implement destructors by implementing the trait `Drop`. This trait requires a unique method (`drop`), and the compiler automatically adds a call to that method when the corresponding value is about to be destroyed.

A.2 Concurrency

The ownership-based approach of Rust makes it very suitable to write concurrent programs.

Shared memory. According to the rules for pointers, in Rust, memory can not be shared. However, the standard library provides ways to allow sharing.

First, the `Mutex` object protects concurrent accesses to the same memory area. This object provides a method `fn lock(&self) -> LockResult<MutexGuard<T>>`. In a nutshell, the returned value (of type `LockResult<MutexGuard<T>>`) gives a *mutable* access to the embedded value (of type `T`). The important point is that the function takes a `&self` reference, i.e. it converts a immutable reference to a mutex to the mutable data it contains[1].

The second point is that multiple threads need to share ownership of the mutex. This can be done using regular `&Mutex<T>` references, but is quite unusual, due to the constraints on `&`-references. Instead, we usually use reference counted pointers, which dynamically verifies that no two pointers attempt to mutate the inner value. In our case, we want to exchange those references across threads, and therefore, we use `Arc<T>`, where `Arc` stands for *Atomically Reference Counted*[2].

Notice that `Mutex` and `Arc` do *not* serve the same purpose: `Arc` allows to share ownership, and `Mutex` provides thread safety. Therefore, to make a type `T` thread safe, we often use `Arc<Mutex<T>>`.

A final note on `Mutex<T>`: if the lock is explicit (the method `lock` explained above), the release is implicit. As explained above, `lock` returns a `LockResult<MutexGuard<T>>`. Hence, if a call to `lock` succeeds, the programmer has access to a `MutexGuard`. To release the lock, the programmer simply `drop`s that guard (`MutexGuard` has an ad-hoc implementation of `Drop`, which transparently releases the lock). With such mechanism, and the ownership rules, there can not be accidental accesses to the object protected by the mutex (if there is an access, it means that the guard has not been dropped, i.e. the mutex was not released).

Message-passing in Rust. To have a message passing model of concurrency in Rust, the standard library provides communication channels, in the `std::sync::mpsc` library. In particular, it provides the function `channel<T>() -> (Sender<T>, Receiver<T>)`, which is used to instantiate `Sender` and `Receiver` structures, which implement *multiple producer, single consumer* message passing primitives.

`Sender<T>` is clonable (hence the multiple producers) and provides a method `send(&self, t: T)` used to send data across threads. It counterpart, `Receiver<T>` implements `recv(&self) -> Result<T, RecvError>` which is used to receive data.

[1] This seems undoable, and actually it is in plain Rust. However, Rust provides a mechanism to temporarily suspend ownership checking, called *unsafe* Rust. In unsafe Rust, the compiler does not provide the usual guarantees, and it is the programmer's job to ensure that what is performed is correct. In the case of a mutex, the overall invariant (no-two simultaneous mutable accesses) is preserved, and checked at runtime.

[2] Again, this behaviour is not doable using only safe Rust. However, with a careful mix of unsafe, `Clone`, and `Drop`, we can achieve the result.

Not all values can be exchanged on such channels. There is a special trait `Send`, inferred by the compiler, which states that a type can safely be exchanged across threads[3].

[3] Almost all *normal* types implement `Send`. The reader can consider that this is not a problem for us, it only requires us to specify it.

Appendix B

Internal Simulation

In this appendix, we illustrate why the second approach to compare location graphs belonging to different models $\mathbb{G}_1$ and $\mathbb{G}_2$ does not work, at least naively. Our goal is to exhibit a systematic method to find a common model $\mathbb{G}$ which is analogous to both $\mathbb{G}_1$ and $\mathbb{G}_2$.

Notice here that our goal is to compare two different models, not to have a location graph that includes locations of both the first and the second model.

We are given two location graphs models $\mathbb{G}_1 = \texttt{lgraph}(\mathbb{P}_1, \mathbb{S}_1, \mathbb{R}_1)$ and $\mathbb{G}_2 = \texttt{lgraph}(\mathbb{P}_2, \mathbb{S}_2, \mathbb{R}_2)$, two sets of unconstrained location transitions $\mathcal{T}_1 = \texttt{trans}(\mathbb{G}_1, \mathbb{A}_1)$ and $\mathcal{T}_2 = \texttt{trans}(\mathbb{G}_2, \mathbb{A}_2)$ (for $\mathbb{A}_1$ and $\mathbb{A}_2$ two sets of labels suitable, using values in $\mathbb{V}_1$ and $\mathbb{V}_2$).

We are given a set $\mathbb{\Delta} = \{\top, \bot\}$ of special symbols.

The merge operator $[\cdot] \star [\cdot]$**.** We now define the operation that merge location graph models, noted $\mathbb{G}_1 \star \mathbb{G}_2$: $\mathbb{G}_1 \star \mathbb{G}_2 = \texttt{lgraph}(\mathbb{P}_1 \cup \mathbb{P}_2, \mathbb{\Delta} \times (\mathbb{S}_1 \cup \mathbb{S}_2), \mathbb{R}_1 \cup \mathbb{R}_2)$.

Before describing the unconstrained transitions, we define a function that lift locations of $\mathbb{G}_1$ (resp. $\mathbb{G}_2$) to locations of $\mathbb{G}_1 \star \mathbb{G}_2$:

Definition 118 (Location lifting function)**.** Given a location $L = [P : s \lhd p \bullet r]$ of a model $\mathbb{G}$, the location $L^\top$ is:

$$[P : \langle \top, s \rangle \lhd p \bullet r]$$

We define $L^\bot$ in an analogous manner.

By extension, we also define the lifting of location graphs, skeleton locations and skeleton location graphs.

Definition 119 (Graph lifting function)**.** Given a graph G,

$$G^\top = \begin{cases} \emptyset & \text{if } G = \emptyset \\ L^\top & \text{if } G = L \\ G_1^\top \parallel G_2^\top & \text{if } G = G_1 \parallel G_2 \end{cases}$$

We define $G^\bot$ in an analogous manner.

Definition 120 (Skeleton location lifting function)**.** Given a skeleton location $L_s = [s \lhd p \bullet r]$ of a model $\mathbb{G}$, the skeleton location $L_s^\top$ is:

$$[\langle \top, s \rangle \lhd p \bullet r]$$

We define $L_s^\bot$ in an analogous manner.

Definition 121 (Skeleton graph lifting function)**.** Given a skeleton graph G_s,

$$G_s^\top = \begin{cases} \emptyset & \text{if } G_s = \emptyset \\ L_s^\top & \text{if } G_s = L_s \\ G_{s1}^\top \parallel G_{s2}^\top & \text{if } G_s = G_{s1} \parallel G_{s2} \end{cases}$$

We define $G_s^\bot$ in an analogous manner.

Notice, of course, that the two lifting functions (for locations and graphs) are sound with respect to the skeleton variants:

$$\Sigma(G)^\top = \Sigma(G^\top) \qquad\qquad \Sigma(L)^\top = \Sigma(L^\top)$$

Well formedness of a merged graph. We now have a location graph model that unite both $\mathbb{G}_1$ and $\mathbb{G}_2$ models. However, we don't want an instance of this model to be composed of instances of both models at the same time. We hence define a well formedness property which asserts that we have only $\top$ or $\bot$ in the graph. This property uses a computation of the *side* of a graph, that is whether the merge graph is built from a graph in $\mathbb{G}_1$ or $\mathbb{G}_2$.

Definition 122 (Side of a merged graph)**.**

$$\texttt{side}(G) \triangleq \begin{cases} \{\top, \bot\} & \text{if } G = \emptyset \\ \{\top\} & \text{if } G = [P : \langle \top, s \rangle \lhd p \bullet r] \\ \{\bot\} & \text{if } G = [P : \langle \bot, s \rangle \lhd p \bullet r] \\ \texttt{side}(G_1) \cap \texttt{side}(G_2) & \text{if } G = G_1 \parallel G_2 \end{cases}$$

Definition 123 (Well-formed merged graph)**.**

$$\texttt{WF}_\star(G) \Leftrightarrow \texttt{side}(G) \neq \emptyset$$

Lemma 34. $\forall G_1, G_2 \in \mathbb{G}_1 \star \mathbb{G}_2 \cdot \texttt{WF}_\star(G_1 \parallel G_2) \Rightarrow (\texttt{WF}_\star(G_1) \wedge \texttt{WF}_\star(G_2))$

Proof.

$$\begin{aligned} \texttt{WF}_\star(G_1 \parallel G_2) &\overset{\text{Def 123}}{\Leftrightarrow} \texttt{side}(G_1 \parallel G_2) \neq \emptyset \\ &\overset{\text{Def 122}}{\Leftrightarrow} \texttt{side}(G_1) \cap \texttt{side}(G_2) \neq \emptyset \\ &\overset{\text{Set theory}}{\Rightarrow} \texttt{side}(G_1) \neq \emptyset \wedge \texttt{side}(G_2) \neq \emptyset \\ &\overset{\text{Def 123}}{\Leftrightarrow} \texttt{WF}_\star(G_1) \wedge \texttt{WF}_\star(G_2) \end{aligned}$$

□

Semantics of merged graphs.

Definition 124 (Unconstrained transitions of a merged location graph)**.** The unconstrained transitions set $\mathcal{T}_{\mathbb{G}_1 \star \mathbb{G}_2}$ of a merged location graph model $\mathbb{G}_1 \star \mathbb{G}_2$ is the smallest set such that:

(i) if $\Delta \cdot \emptyset \rhd L \xrightarrow{\Lambda} G \in \mathcal{T}_1$, then $\Delta \cdot \emptyset \rhd L^\top \xrightarrow{\Lambda} G^\top \in \mathcal{T}_{\mathbb{G}_1 \star \mathbb{G}_2}$

(ii) if $\Delta \cdot \emptyset \rhd L \xrightarrow{\Lambda} G \in \mathcal{T}_2$, then $\Delta \cdot \emptyset \rhd L^\bot \xrightarrow{\Lambda} G^\bot \in \mathcal{T}_{\mathbb{G}_1 \star \mathbb{G}_2}$

where $\mathcal{T}_1$ (resp. $\mathcal{T}_2$) is the set of unconstrained transition of $\mathbb{G}_1$ (resp. $\mathbb{G}_2$).

Notice that, for any rule $\Gamma \rhd L \xrightarrow{\Lambda} G \in \mathcal{T}_{\mathbb{G}_1 \star \mathbb{G}_2}$, $\texttt{side}(L) = \texttt{side}(G)$. Thus, the well-formness is preserved through reduction:

The authorisation function $\texttt{Auth}_\star$ lifts $\texttt{Auth}_1$ and $\texttt{Auth}_2$:

Definition 125 (Lifted authorisation function)**.**

$$\texttt{Auth}_\star(G_s^\top, \Delta \cdot \emptyset \rhd L^\top \xrightarrow{\Lambda} C^\top) \triangleq \texttt{Auth}_1(G_s, \Delta \cdot \emptyset \rhd L \xrightarrow{\Lambda} C)$$

$$\texttt{Auth}_\star(G_s^\bot, \Delta \cdot \emptyset \rhd L^\bot \xrightarrow{\Lambda} C^\bot) \triangleq \texttt{Auth}_2(G_s, \Delta \cdot \emptyset \rhd L \xrightarrow{\Lambda} C)$$

and $\texttt{Auth}_\star$ is not defined when mixing $\top$ and $\bot$.

Unfortunatelly, with such naive approach, we can not apply the standard definition of simulation directly. Indeed, the standard definition requires a simulation $\mathcal{R}$ that:

$$\forall \Gamma \in \mathbb{A} \times \mathbb{G}^{\mathsf{s}} \cdot \Gamma \cup \Sigma(C) \vdash_{\mathcal{T}_u} C \xrightarrow{\Lambda} C' \Rightarrow \Gamma \cup \Sigma(D) \vdash_{\mathcal{T}_u} D \xrightarrow{\Lambda'} D'$$

with $\langle C', D' \rangle \in \mathcal{R}$ and $\Lambda \equiv \Lambda'$. However, $\Gamma \cup \Sigma(C)$ and $\Gamma \cup \Sigma(D)$ do not have the same side (at least one of them mixes two sides), and therefore, one of them can not reduce, according to our authorisation function.

Bibliography

[1] *Akka: build concurrent, distributed, and resilient message-driven applications for Java and Scala — Akka*. URL: https://akka.io/ (visited on 03/09/2020) (cit. on pp. 4, 109).

[2] Jonathan Aldrich and Craig Chambers. "Ownership Domains: Separating Aliasing Policy from Mechanism". en. In: *ECOOP 2004 – Object-Oriented Programming*. Ed. by Martin Odersky. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2004, pp. 1–25. ISBN: 978-3-540-24851-4. DOI: 10.1007/978-3-540-24851-4_1 (cit. on pp. 15, 16).

[3] Paulo Sérgio Almeida. "Balloon types: Controlling sharing of state in data types". en. In: *ECOOP'97 — Object-Oriented Programming*. Ed. by Mehmet Akşit and Satoshi Matsuoka. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1997, pp. 32–59. ISBN: 978-3-540-69127-3. DOI: 10.1007/BFb0053373 (cit. on p. 15).

[4] Frederico Alvares De Oliveira Jr., Eric Rutten, and Lionel Seinturier. *High-level Language Support for the Control of Reconfiguration in Component-based Architectures*. Research Report RR-8669. INRIA Grenoble - Rhône-Alpes ; INRIA Lille - Nord Europe ; Laboratoire d'Informatique Fondamentale de Lille ; INRIA, Jan. 2015. URL: https://hal.inria.fr/hal-01103548 (cit. on p. 4).

[5] Frederico Alvares De Oliveira Jr., Eric Rutten, and Lionel Seinturier. "High-level Language Support for the Control of Reconfigurations in Component-based Architectures". In: *9th European Conference on Software Architecture (ECSA)*. Vol. 9278. LNCS. Backup Publisher: Danny weyns and Raffaela Mirandola and Ivica Crnkovic. Dubrovnick, Croatia: Springer, Sept. 2015, pp. 285–293. URL: https://hal.inria.fr/hal-01160612 (cit. on p. 4).

[6] Vytautas Astrauskas et al. "Leveraging rust types for modular specification and verification". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), 147:1–147:30. DOI: 10.1145/3360573. URL: https://doi.org/10.1145/3360573 (visited on 12/03/2020) (cit. on p. 157).

[7] Ananda Basu, Marius Bozga, and Joseph Sifakis. "Modeling Heterogeneous Real-time Components in BIP". In: *Fourth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2006), 11-15 September 2006, Pune, India*. IEEE Computer Society, 2006, pp. 3–12 (cit. on p. 24).

[8] Philippe Bidinger and Jean-Bernard Stefani. "The Kell Calculus: Operational Semantics and Type System". en. In: *Formal Methods for Open Object-Based Distributed Systems*. Ed. by Elie Najm, Uwe Nestmann, and Perdita Stevens. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2003, pp. 109–123. ISBN: 978-3-540-39958-2. DOI: 10.1007/978-3-540-39958-2_8 (cit. on p. 11).

[9] Simon Bliudze and Joseph Sifakis. "The algebra of connectors: structuring interaction in BIP". In: *Proceedings of the 7th ACM & IEEE International conference on Embedded software, EMSOFT 2007, September 30 - October 3, 2007, Salzburg, Austria*. ACM, 2007, pp. 11–20 (cit. on p. 24).

[10] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. "Access control for mobile agents: The calculus of boxed ambients". In: *ACM Transactions on Programming Languages and Systems* 26.1 (Jan. 2004), pp. 57–124. ISSN: 0164-0925. DOI: 10.1145/963778.963781. URL: https://doi.org/10.1145/963778.963781 (visited on 01/26/2021) (cit. on pp. 10, 23).

[11] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. "Mobility Types for Mobile Ambients". en. In: *Automata, Languages and Programming*. Ed. by Gerhard Goos et al. Vol. 1644. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 230–239. ISBN: 978-3-540-66224-2 978-3-540-48523-0. DOI: 10.1007/3-540-48523-6_20. URL: http://link.springer.com/10.1007/3-540-48523-6_20 (visited on 01/28/2021) (cit. on p. 10).

[12] Luca Cardelli and Andrew D. Gordon. "Mobile ambients". en. In: *Foundations of Software Science and Computation Structures*. Ed. by Maurice Nivat. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1998, pp. 140–155. ISBN: 978-3-540-69720-6. DOI: 10.1007/BFb0053547 (cit. on pp. 10, 23).

[13] Ilaria Castellani. "CHAPTER 15 - Process Algebras with Localities". en. In: *Handbook of Process Algebra*. Ed. by J. A. Bergstra, A. Ponse, and S. A. Smolka. Amsterdam: Elsevier Science, Jan. 2001, pp. 945–1045. ISBN: 978-0-444-82830-9. DOI: 10.1016/B978-044482830-9/50033-3. URL: http://www.sciencedirect.com/science/article/pii/B9780444828309500333 (visited on 01/21/2021) (cit. on p. 23).

[14] Dominik Charousset, Raphael Hiesgen, and Thomas C. Schmidt. "Revisiting actor programming in C++". en. In: *Computer Languages, Systems & Structures* 45 (Apr. 2016), pp. 105–131. ISSN: 1477-8424. DOI: 10.1016/j.cl.2016.01.002. URL: http://www.sciencedirect.com/science/article/pii/S1477842416000038 (visited on 07/23/2020) (cit. on p. 4).

[15] Dave Clarke and Tobias Wrigstad. "External Uniqueness". In: *In Workshop on Foundations of Object-Oriented Languages (FOOL*. 2003 (cit. on p. 15).

[16] Dave Clarke et al. "Ownership Types: A Survey". In: *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*. Ed. by David Hutchison et al. Vol. 7850. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 15–58. ISBN: 978-3-642-36945-2 978-3-642-36946-9. URL: http://link.springer.com/10.1007/978-3-642-36946-9_3 (visited on 03/02/2017) (cit. on p. 11).

[17] David G. Clarke, John M. Potter, and James Noble. "Ownership Types for Flexible Alias Protection". In: *Proceedings of the 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. OOPSLA '98. event-place: Vancouver, British Columbia, Canada. New York, NY, USA: ACM, 1998, pp. 48–64. ISBN: 1-58113-005-8. DOI: 10.1145/286936.286947. URL: http://doi.acm.org/10.1145/286936.286947 (cit. on pp. 11–13).

[18] Joeri De Koster et al. "Domains: Safe sharing among actors". en. In: *Science of Computer Programming* 98 (Feb. 2015), pp. 140–158. ISSN: 01676423. DOI: 10.1016/j.scico.2014.02.008. URL: https://linkinghub.elsevier.com/retrieve/pii/S0167642314000495 (visited on 01/28/2021) (cit. on p. 12).

[19] R. De Nicola, G.L. Ferrari, and R. Pugliese. "KLAIM: a kernel language for agents interaction and mobility". en. In: *IEEE Transactions on Software Engineering* 24.5 (May 1998), pp. 315–330. ISSN: 00985589. DOI: 10.1109/32.685256. URL: http://ieeexplore.ieee.org/document/685256/ (visited on 01/28/2021) (cit. on pp. 10, 11).

[20] Rocco De Nicola. "From Process Calculi to Klaim and Back". en. In: *Electronic Notes in Theoretical Computer Science*. Proceedings of the Workshop "Essays on Algebraic Process Calculi" (APC 25) 162 (Sept. 2006), pp. 159–162. ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.12.110. URL: http://www.sciencedirect.com/science/article/pii/S1571066106004269 (visited on 01/28/2021) (cit. on p. 10).

[21] Pierpaolo Degano and Ugo Montanari. "A model for distributed systems based on graph rewriting". In: *Journal of the ACM* 34.2 (Apr. 1987), pp. 411–449. ISSN: 0004-5411. DOI: 10.1145/23005.24038. URL: https://doi.org/10.1145/23005.24038 (visited on 01/27/2021) (cit. on p. 24).

[22] Dominique Devriese, Lars Birkedal, and Frank Piessens. "Reasoning about Object Capabilities with Logical Relations and Effect Parametricity". en. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbrucken: IEEE, Mar. 2016, pp. 147–162. ISBN: 978-1-5090-1751-5 978-1-5090-1752-2. DOI: 10.1109/EuroSP.2016.22. URL: http://ieeexplore.ieee.org/document/7467352/ (visited on 04/15/2021) (cit. on p. 19).

[23] J. Eker et al. "Taming heterogeneity - the Ptolemy approach". In: *Proceedings of the IEEE* 91.1 (Jan. 2003). Conference Name: Proceedings of the IEEE, pp. 127–144. ISSN: 1558-2256. DOI: 10.1109/JPROC.2002.805829 (cit. on p. 24).

[24] *Erlang Programming Language*. URL: https://www.erlang.org/ (visited on 12/03/2020) (cit. on p. 158).

[25] David Garlan, Robert T. Monroe, and David Wile. "Acme: Architectural Description of Component-Based Systems". In: *Foundations of Component-Based Systems*. Ed. by Gary T. Leavens and Murali Sitaraman. Section: 3. New York, NY: Cambridge University Press, 2000, pp. 47–67. ISBN: 0-521-77164-1 (cit. on pp. 2, 3).

[26] David Gelernter. “Generative communication in Linda”. en. In: *ACM Transactions on Programming Languages and Systems* 7.1 (Jan. 1985), pp. 80–112. ISSN: 0164-0925, 1558-4593. DOI: 10.1145/2363.2433. URL: https://dl.acm.org/doi/10.1145/2363.2433 (visited on 01/28/2021) (cit. on p. 10).

[27] Robin Gibaud. “Application of the Discrete Element Method to Finite Inelastic Strain in Multi-Materials”. Theses. Université Grenoble Alpes, Nov. 2017. URL: https://tel.archives-ouvertes.fr/tel-01761756 (cit. on p. ii).

[28] Matthew Hennessy. *A Distributed Pi-Calculus*. USA: Cambridge University Press, 2007. ISBN: 978-0-521-87330-7 (cit. on p. 10).

[29] Matthew Hennessy and James Riely. “Resource Access Control in Systems of Mobile Agents”. en. In: *Information and Computation* 173.1 (Feb. 2002), pp. 82–120. ISSN: 0890-5401. DOI: 10.1006/inco.2001.3089. URL: http://www.sciencedirect.com/science/article/pii/S0890540101930895 (visited on 01/29/2021) (cit. on p. 10).

[30] Matthew Hennessy and James Riely. “Resource Access Control in Systems of Mobile Agents: (Extended Abstract)”. en. In: *Electronic Notes in Theoretical Computer Science*. HLCL ’98, 3rd International Workshop on High-Level Concurrent Languages (Satellite Workshop of CONCUR ’98) 16.3 (Jan. 1998), pp. 174–188. ISSN: 1571-0661. DOI: 10.1016/S1571-0661(04)00141-0. URL: http://www.sciencedirect.com/science/article/pii/S1571066104001410 (visited on 01/29/2021) (cit. on p. 10).

[31] Carl Hewitt, Peter Bishop, and Richard Steiger. “A universal modular ACTOR formalism for artificial intelligence”. In: *Proceedings of the 3rd international joint conference on Artificial intelligence*. IJCAI’73. Stanford, USA: Morgan Kaufmann Publishers Inc., Aug. 1973, pp. 235–245. (Visited on 07/23/2020) (cit. on pp. 2, 4, 12).

[32] John Hogg. “Islands: aliasing protection in object-oriented languages”. In: *ACM SIGPLAN Notices* 26.11 (Nov. 1991), pp. 271–285. ISSN: 0362-1340. DOI: 10.1145/118014.117975. URL: https://doi.org/10.1145/118014.117975 (visited on 08/04/2020) (cit. on p. 15).

[33] Jean-Bernard Stefani and Martin Vassor. *The Hypercell Framework* (cit. on pp. 23, 26, 37, 39, 40, 59).

[34] Ivan Lanese and Emilio Tuosto. “Synchronized Hyperedge Replacement for Heterogeneous Systems”. en. In: *Coordination Models and Languages*. Ed. by Jean-Marie Jacquet and Gian Pietro Picco. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2005, pp. 220–235. ISBN: 978-3-540-32006-7. DOI: 10.1007/11417019_15 (cit. on p. 24).

[35] Francesca Levi and Davide Sangiorgi. “Mobile safe ambients”. In: *ACM Transactions on Programming Languages and Systems* 25.1 (Jan. 2003), pp. 1–69. ISSN: 0164-0925. DOI: 10.1145/596980.596981. URL: https://doi.org/10.1145/596980.596981 (visited on 01/26/2021) (cit. on pp. 10, 23).

[36] Pascal Maerki. “Flexible Dynamic Ownership in Smalltalk”. English. Bachelor Thesis. Bern: Universität Bern, Feb. 2013. URL: http://scg.unibe.ch/archive/projects/Maer13a-DynamicOwnership.pdf (cit. on p. 12).

[37] Mark Samuel Miller. “Towards a Unified Approach to Access Control and Concurrency Control”. en. PhD thesis. John Hopkins University, 2006 (cit. on p. 19).

[38] R. Milner. *The space and motion of communicating agents*. OCLC: ocn261177529. Cambridge : New York: Cambridge University Press, 2009. ISBN: 978-0-521-49030-6 978-0-521-73833-0 (cit. on p. 9).

[39] Robin Milner. *Communicating and mobile systems: the π-calculus*. eng. Cambridge: Cambridge Univ. Press, 1999. ISBN: 978-0-521-65869-0 978-0-521-64320-7 (cit. on p. 9).

[40] Robin Milner. “The Polyadic π-Calculus: a Tutorial”. In: *Logic and Algebra of Specification*. Ed. by Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 203–246. ISBN: 978-3-642-58041-3 (cit. on p. 9).

[41] Naftaly H. Minsky. “Towards alias-free pointers”. en. In: *ECOOP ’96 — Object-Oriented Programming*. Ed. by Pierre Cointe. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1996, pp. 189–209. ISBN: 978-3-540-68570-8. DOI: 10.1007/BFb0053062 (cit. on p. 15).

[42] Naftaly H. Minsky and Victoria Ungureanu. "Law-governed interaction: a coordination and control mechanism for heterogeneous distributed systems". In: *ACM Transactions on Software Engineering and Methodology* 9.3 (July 2000), pp. 273–305. ISSN: 1049-331X. DOI: 10.1145/352591.352592. URL: https://doi.org/10.1145/352591.352592 (visited on 08/05/2020) (cit. on pp. 20, 77).

[43] Johan Östlund and Tobias Wrigstad. "Multiple Aggregate Entry Points for Ownership Types". In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by David Hutchison et al. Vol. 7313. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 156–180. ISBN: 978-3-642-31056-0 978-3-642-31057-7. URL: http://link.springer.com/10.1007/978-3-642-31057-7_8 (visited on 03/09/2017) (cit. on pp. 5, 12, 14).

[44] D. L. Parnas. "On the criteria to be used in decomposing systems into modules". In: *Communications of the ACM* 15.12 (Dec. 1972), pp. 1053–1058. ISSN: 0001-0782. DOI: 10.1145/361598.361623. URL: https://doi.org/10.1145/361598.361623 (visited on 01/25/2021) (cit. on p. 1).

[45] Jonathan Protzenko. "Mezzo: a typed language for safe effectful concurrent programs". en. PhD thesis. Université Paris Diderot - Paris 7, Sept. 2014. URL: https://hal.inria.fr/tel-01086106 (visited on 02/01/2021) (cit. on pp. 19, 20).

[46] Claudius Ptolemaeus, ed. *System design, modeling, and simulation: using Ptolemy II*. en. 1. ed., version 1.02. OCLC: 935837595. Berkeley, Calif: UC Berkeley EECS Dept, 2014. ISBN: 978-1-304-42106-7 (cit. on p. 24).

[47] *Rust Programming Language*. en-US. Library Catalog: www.rust-lang.org. URL: https://www.rust-lang.org/ (visited on 03/09/2020) (cit. on p. 89).

[48] *Rust's powerful actor system and most fun web framework*. URL: https://actix.rs/ (visited on 07/23/2020) (cit. on p. 4).

[49] J. J. M. M. Rutten. "Coalgebra, Concurrency, and Control". In: *Proceedings of the 5th Workshop on Discrete Event Systems (wodes 2000*. 1999, pp. 31–38 (cit. on pp. 40, 42).

[50] Quentin Sabah. "Siaam : Simple Isolation for an Actor-based Abstract Machine". Issue: 2013GRENM082. Theses. Université de Grenoble, Dec. 2013. URL: https://tel.archives-ouvertes.fr/tel-01560981 (cit. on pp. 12, 116).

[51] Davide Sangiorgi. *Introduction to Bisimulation and Coinduction*. Cambridge: Cambridge University Press, 2011. ISBN: 978-1-107-00363-7. DOI: 10.1017/CBO9780511777110. URL: https://www.cambridge.org/core/books/introduction-to-bisimulation-and-coinduction/8B54001CB763BAE9C4BA602C0A341D60 (visited on 08/27/2020) (cit. on p. 39).

[52] Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. "Environmental bisimulations for higher-order languages". In: *ACM Transactions on Programming Languages and Systems* 33.1 (Jan. 2011), 5:1–5:69. ISSN: 0164-0925. DOI: 10.1145/1889997.1890002. URL: https://doi.org/10.1145/1889997.1890002 (visited on 10/07/2020) (cit. on p. 42).

[53] Davide Sangiorgi and Jan Rutten, eds. *Advanced Topics in Bisimulation and Coinduction*. Cambridge Tracts in Theoretical Computer Science. Cambridge: Cambridge University Press, 2011. ISBN: 978-1-107-00497-9. DOI: 10.1017/CBO9780511792588. URL: https://www.cambridge.org/core/books/advanced-topics-in-bisimulation-and-coinduction/A7949A3E2CD7A9C365365ED0F81D6C77 (visited on 08/27/2020) (cit. on p. 39).

[54] Alan Schmitt and Jean-Bernard Stefani. "The Kell Calculus: A Family of Higher-Order Distributed Process Calculi". en. In: *Global Computing*. Ed. by Corrado Priami and Paola Quaglia. Vol. 3267. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 146–178. ISBN: 978-3-540-24101-0 978-3-540-31794-4. DOI: 10.1007/978-3-540-31794-4_9. URL: http://link.springer.com/10.1007/978-3-540-31794-4_9 (visited on 01/29/2021) (cit. on p. 11).

[55] Constantin Serban and Naftaly Minsky. "Generalized Access Control of Synchronous Communication". en. In: *Middleware 2006*. Ed. by Maarten van Steen and Michi Henning. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006, pp. 281–300. ISBN: 978-3-540-68256-1. DOI: 10.1007/11925071_15 (cit. on p. 20).

[56] Constantin Serban and Naftaly Minsky. "In Vivo Evolution of Policies that Govern a Distributed System". In: *2009 IEEE International Symposium on Policies for Distributed Systems and Networks*. July 2009, pp. 134–141. DOI: 10.1109/POLICY.2009.25 (cit. on p. 20).

[57] Jean-Bernard Stefani and Martin Vassor. "Encapsulation and Sharing in Dynamic Software Architectures: The Hypercell Framework". In: *Formal Techniques for Distributed Objects, Components, and Systems*. Ed. by Jorge A. Pérez and Nobuko Yoshida. Springer International Publishing, 2019, pp. 242–260. ISBN: 978-3-030-21759-4 (cit. on pp. 7, 23, 39).

[58] David Swasey, Deepak Garg, and Derek Dreyer. "Robust and Compositional Verification of Object Capability Patterns". en. In: *Proceedings of the ACM on Programming Languages* 1 (), p. 26 (cit. on p. 19).

[59] *The Scala Programming Language*. Library Catalog: scala-lang.org. URL: `https://www.scala-lang.org/` (visited on 03/09/2020) (cit. on p. 109).

[60] Janina Voigt. "Access contracts: a dynamic approach to object-oriented access protection". PhD Thesis. University of Cambridge, UK, 2015. URL: `http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.708888` (cit. on p. 15).

[61] Janina Voigt. *Access contracts: a dynamic approach to object-oriented access protection*. Tech. rep. UCAM-CL-TR-880. University of Cambridge, Computer Laboratory, Feb. 2016. URL: `https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-880.pdf` (cit. on pp. 15, 18).

[62] Erwann Wernli, Pascal Maerki, and Oscar Nierstrasz. "Ownership, Filters and Crossing Handlers: Flexible Ownership in Dynamic Languages". In: *SIGPLAN Not.* 48.2 (Oct. 2012), pp. 83–94. ISSN: 0362-1340. DOI: `10.1145/2480360.2384589`. URL: `http://doi.acm.org/10.1145/2480360.2384589` (cit. on p. 12).

Index

Abstract Machine
  List of Transitions
    Identified, 102
  Location, 99
  Location Graph
    Well-Formed, 106
  Picking, 102
  Primitives, 98
  Role predicate, 103
  Transition, 95
    Complete Set, 102
    Endpoint Effects, 97
    Identified, 102
    Messages, 97
    New Sort, 97
    Result, 97
    Skeleton, 96
    Sum, 95
    Well-formed, 96
  Transition Application
    Location, 105
  Transition application
    Full, 104
    Skeleton Location, 104
    Skeleton Location Graph, 104
Authorisation function, 32
  2^{nd} order, 55

Bisimulation
  Partial, 42

Encapsulation Policy, 3
Endpoint, 94
Environment, 30
  Correct, 33
  Elements, 30
  Union, 30
Exchange, 103

Graph partition
  Coarser, 47
  Finer, 47

Interaction, 31
  2^{nd} order, 53
  Independent, 34
  Matching, 33
    Union, 33
Isolation Policy, 3

Label, 31
  Elements, 31
  Graph Recovery, 55
  Interactions
    Elements, 31
  Priority constraint
    Elements, 31
  Pruning, 55
Label range, 76
Lifting function
  Graph, 163
  Location, 163
  Skeleton graph, 164
  Skeleton location, 163
Location, 25
  Handle, 98
    Well-formed, 99
  Nesting, 45
    Inverse, 46
Location Graph, 26
  Σ, 27
  Elements, 25
  Inclusion, 38
  Locations, 38
  Nesting, 49
    Inverse, 50
  Partition, 46
  Partitioning Function, 46
  Size, 37
Location Pregraph, 25
  Separated, 26
  Structural Equivalence, 26
  Well formed, 26

Nesting function
  Graph
    Recursive, 67
  Higher-order, 68
  Location
    Recursive, 67

Ownership domain
  Strict, 75
    Excluding owner, 75

Partitioning function
  Finer, 49
Prelocation, 25
  Elements, 25
  Well formed, 25
Priority

Satisfaction, 32
Composition, 33
Priority Constraint, 30

Simulation
Heterogenous, 41
Strong, 40
Skeleton
Extraction, 54
Skeleton Location Graph, 27
Elements, 27
Inclusion, 29
Structural Equivalence, 28
Union, 29
Skeleton location graphs(Abstract machine), 94
Skeleton locations (Abstract machine), 94
Strict encapsulation
Partitioning function, 78
Support, 27

Transition
Item, 95
Skeleton, 96
Sum, 95
List
Result, 105
Transition item
Enqueuing, 99

Unconstrained location transition, 32