# Analyse des systèmes mobiles par interprétation abstraite.

Jérôme Feret

# THÈSE

présentée à

l'ÉCOLE POLYTECHNIQUE

pour l'obtention du titre de

**DOCTEUR DE L'ÉCOLE POLYTECHNIQUE EN INFORMATIQUE**

**Jérôme Feret**

25 février 2005

# Analyse des systèmes mobiles par interprétation abstraite

*Analysis of mobile systems by abstract interpretation*

| | |
|---|---|
| **Directeur de thèse:** | Patrick Cousot |
| | *Professeur, École Normale Supérieure, Paris* |
| **Président:** | M. David Schmidt |
| | *Professeur, Kansas State University* |
| **Rapporteurs:** | M. Luca Cardelli |
| | *Directeur assistant, Microsoft Research, Cambridge* |
| | M. David Schmidt |
| | *Professeur, Kansas State University* |
| **Examinateurs:** | M. Vincent Danos |
| | *Directeur de recherches au* CNRS, *Paris VII* |
| | M. Roberto Giacobazzi |
| | *Professeur, Università degli Studi di Verona* |
| | M. Jean Goubault-Larrecq |
| | *Professeur associé, École Normale Supérieure de Cachan* |

Cette recherche a été conduite à l'École Normale Supérieure (Paris) durant un contrat d'allocation couplée (normalien) de l'Université Paris IX Dauphine. Cette recherche a été financée en partie par le projet DAEDALUS (projet européen IST-1999-20527 du programme FP5).

## Résumé

Un système mobile est un ensemble de composants qui peuvent interagir entre eux, tout en modifiant dynamiquement le système lui-même. Ces interactions contrôlent ainsi la création et la destruction des liaisons entre les composants, mais aussi la création dynamique de nouveaux composants au sein du système. La taille d'un tel système varie au cours du temps, elle n'est pas bornée en général. Un système mobile peut représenter des réseaux de télécommunication, des systèmes reconfigurables, des applications *client-serveur* sur la toile, des protocoles cryptographiques, ou des systèmes biologiques. Plusieurs modèles sont disponibles selon le domaine d'application et la granularité du niveau d'observation.

Dans cette thèse, nous proposons un cadre de travail unifiant pour découvrir et prouver statiquement (avant leur exécution) et automatiquement les propriétés des systèmes mobiles. Nous proposons un méta-langage dans lequel nous encodons les modèles les plus couramment utilisés dans la littérature (le $\pi$-calcul, le calcul des *ambients*, le *join*-calcul, le *spi*-calcul, les BIO-*ambients*, etc). Pour chaque modèle encodé, le méta-langage calcule une sémantique enrichie dans laquelle à la fois les composants et les objets qu'ils manipulent (adresses mémoires, noms de canaux, clefs secrètes ou partagées, etc) sont identifiés par l'historique de leur création. Ainsi, nous n'utilisons pas de relation de congruence (ni de renommage), ce qui rend l'analyse plus facile.

Le cadre général de l'Interprétation Abstraite nous permet ensuite de dériver des sémantiques abstraites, qui sont décidables, correctes, et approchées. Dans cette thèse, nous donnons trois analyses génériques que nous instancions selon le compromis désiré entre le temps de calcul et la précision de l'analyse. La première analyse se concentre sur les propriétés dynamiques du système. Elle infère des relations entre les historiques des objets qui sont manipulés par les composants du système. Cette analyse distingue les instances récursives d'un même objet, et ce, même lorsque le nombre de ces instances n'est pas borné. à titre d'exemple, cette analyse prouve dans le cas d'une application *client-serveur* à nombre illimité de clients, que les données de chaque client ne sont pas communiquées aux autres clients. La deuxième analyse se concentre sur des propriétés de concurrence. Cette analyse compte le nombre de composants du système. Elle permet de détecter que certains composants ne peuvent pas interagir, car ils ne coexistent jamais. Elle peut aussi garantir à un système qu'il n'épuisera pas les ressources physiques disponibles. Une troisième analyse mêle concurrence et dynamicité.

## Abstract

A mobile system is a pool of agents that may interact with each other. These interactions dynamically change the system, by controlling both creation and destruction of links between agents. These interactions also control the creation of new agents. The size of a mobile system evolves during its computation. This size may be unbounded. A mobile system may describe telecommunication networks, reconfigurable systems, *client-server* applications, cryptographic protocols, or biological systems. Several models are available according to the application field and the granularity of the observation level.

In this thesis, we propose a unifying framework to discover and prove automatically and statically some properties of mobile systems. We propose a meta-language to encode the most current models for mobility (the $\pi$-calculus, the *ambients*, the *join*-calculus, the *spi*-calculus, the BIO-*ambients*, and so on). The meta-language provides an operational semantics for each encoded model. In these semantics, each agent is identified with the history of its creation, so that this semantics avoids the use of $\alpha$-conversion.

Then, we use the Abstract Interpretation framework to derive abstract semantics, which are sound, decidable, but approximate. In this thesis, we give three generic semantics that we set according to the expected trade-off between accuracy and efficiency. The first analysis focuses on dynamic properties: it captures relations about the creation histories of the agents of the system. This analysis is precise enough to distinguish recursive instances of each agent, even when there is an unbounded number of instances. Thus, we can prove in the case of a *client-sever* application that the server always returns data to the right client. The second analysis focuses on concurrency properties: it counts the number of occurrences of agents inside the system. This analysis detects mutual exclusion and it bounds the number of agents. The third analysis mixes concurrency and dynamic properties. It gathers the agents of the system in several computation unit. Then, it abstract the number of occurrence of agent in each computation unit. For instance, we can prove the absence of race in the specification of a shared-memory with dynamic allocation that is written in the $\pi$-calculus.

## Remerciements

Je remercie Patrick COUSOT pour avoir encadré mon travail au cours de mes années de thèse. J'ai bénéficié de ses conseils avisés. Je remercie également Radhia COUSOT et Arnaud VENET qui m'ont initié aux joies du $\pi$-calcul dans le cadre de mes stages de maîtrise et de DEA. Arnaud VENET m'a fait partagé son enthousiasme et m'a insufflé les bases pour l'écriture d'articles scientifiques.

Je remercie également Antoine MINÉ et Xavier RIVAL, qui partagent mon bureau. Ils m'ont toujours fait part de leurs fines remarques. Je remercie aussi les autres membres de l'équipe : Julien BERTRANE, Bruno BLANCHET, Laurent MAUBORGNE et David MONNIAUX, ainsi que ceux de la «sister team» de l'École polytechnique : Guillaume CAPRON, Charles HYMANS, Francesco LOGOZZO, Damien MASSÉ et Élodie-Jane SIMS. Ils ont tous su me faire profiter au mieux de leur expérience, et se sont toujours proposés dans la tâche, au combien fastidieuse, de la relecture de mes papiers. J'adresse une mention particulière pour Eben UPTON pour sa patiente relecture de mon article pour le numéro spécial de journal. Pierre-Loïc GAROCHE a aussi participé à la relecture de cette thèse. Je remercie aussi Yves VERHOEVEN qui sera toujours le bienvenu pour nous faire partager sa bonne humeur.

Je tiens, par ailleurs, à remercier vivement le groupe de travail sur la concurrence de l'équipe PPS, et tout particulièrement Vincent DANOS et François MAUREL. Ce groupe de travail m'a permis d'élargir le cadre de ma recherche et de faire partager mes travaux à une autre communauté.

Je remercie aussi tous les membres de mon jury de thèse : Ms Luca CARDELLI, Patrick COUSOT, Vincent DANOS, David SCHMIDT, Roberto GIACOBIAZZI et Jean GOUBAULT-LARRECQ, pour l'attention qu'ils ont portée à mes travaux, et les judicieuses remarques qu'ils m'ont faites dans leur rapport et lors de la soutenance.

Le Laboratoire d'Informatique de l'École Normale Supérieure est un cadre agréable pour mener des recherches. Je tiens à remercier Mmes ANGELY, IMBERT, ISNARD, et MONGIAT, et M DENISE pour leur traitement efficace des problèmes administratifs.

Je remercie également toute ma famille. Je remercie en particulier mon père qui m'a initié très jeune aux joies de l'informatique. Enfin, je remercie Natacha SIMIC. Elle a su me prodiguer des conseils capitaux tant pour mon anglais que pour mon éloquence. Elle m'a aussi aidé à rendre mes transparents beaucoup plus impactants. J'ai ainsi pu améliorer mes prestations orales. Elle m'a aussi apporté son soutien quotidien durant cette thèse. Elle m'a aussi écouté, elle a su s'intéresser et suivre de près mes travaux de recherche.

# Table des matières

# Chapter 1

# Introduction

## 1.1 Motivations

### 1.1.1 Mobility models

A mobile system is a pool of threads. These threads may establish some links between each other, and use these links to interact. These interactions control the dynamically creation of both new threads and new links between threads (interactions may also control the updating or the destruction of threads and of these links). This way the interaction topology of mobile systems dynamically changes during computation, making their analysis a very difficult task.

Mobile systems are widely used in several distinct areas: they can describe communicating distributed systems; they are broadly used to model ideal cryptographic protocols abstracting away algorithmic primitives; they are also used now to model biological systems. Moreover, there exist several models according to the granularity of the model abstraction. For instance, in the $\pi$-calculus, two threads may establish links when they know the name of a common channel; while in the ambient calculus, rooting is explicit and links may only be established when spatial conditions are satisfied. As a consequence, there is a growing number of models for mobility, and several separate scientific communities. In this thesis, we propose a unifying approach for analyzing some properties of mobile systems. Our framework can be applied to most frequently used models in the current literature (such as the $\pi$-calculus, the *spi*-calculus, and the *ambient* calculus).

### 1.1.2 Non standard semantics

Mobile models widely use names (channel names, ambient names, memory cell names, etc...). When a thread opens a channel, builds an ambient, or allocates a memory cell, the thread gives it a name. Then, this name may be communicated to the other threads during computations. A name is atomic, which means that it cannot be computed: the only way to get a name is to have received this name from another thread. Dynamic creation of names is a key notion in mobile systems. It allows for the description of unbounded systems using an unbounded number of names. Mobility models usually define their semantics up to a congruence relation that helps in dealing with this unboundedness. Each name is protected by scoping rules. Nevertheless, when two threads interact, they must extrude the scope of their names to each other. The use of $\alpha$-conversion solves conflicts (when two threads have associated the same name to two distinct things). Unfortunately, this makes any analysis quite uneasy. On one hand, we must consider the semantics up to equivalence classes, on the other hand there are no explicit relations between the threads and what they declare. In order to tackle this problem, we introduce a non standard semantics where name allocation is fully deterministic. Each allocated name is computed from the local history of the thread that has opened the channel, built the ambient, or allocated the memory cell. This allows for expressing explicitly that two names have been allocated by the same instance of a thread since it is encoded inside the names themselves. The non standard semantics gets rid of congruence relations: we always extrude the scope of names at the top level and we use thread sets to encode parallel composition.

In this thesis, we propose a generic framework to build non standard semantics. We have proposed a meta language and we have encoded the models most frequently used in the literature. This meta language allows for the description of: matching guards, internal and external choices, guarded replication and explicit recursion, location, migration and dissolution, term construction and destruction, safe migration, and channeled communication across boundaries.

### 1.1.3 Analysis issues

Having chosen the appropriate semantics, we use the Abstract Interpretation framework to design decidable analyses. This framework is highly generic: it can be applied to various analyses, provided some abstract primitives are given. Moreover, it is extensible: it allows us to build the (approximated reduced) product of several analyses expressed in this framework.

Then, we use our framework to address several orthogonal issues:

1. Environment analysis consists in detecting relationships among the thread instances that have allocated some names and the thread instances that receive these names. This analysis is context independent: it detects which names can be communicated outside the system. It is also non-uniform in the sense that it distinguishes between recursive instances of threads. We

can prove, for instance, that a name can be communicated to at most one other instance of a thread, and not to the other ones. In the case of an *ftp*-server, it detects that the server may return a query only to the client which has sent the corresponding request, even in the case of an unbounded number of clients.

2. Occurrence counting analysis consists in abstracting the occurrence number of thread instances during computation sequences. It is especially useful to detect mutual exclusion. It also helps in discovering a bound to the number of agents during computation sequences, so that we can verify that some part of the systems will not exceed physical limits imposed by the implementation of the system. In the case of an *ftp*-server, we can automatically infer the maximum number of simultaneous client sessions. Our approach relies on the use of a reduced product between a non-relational and a relational domain. Complexity problems are solved by using approximated algorithms for computing a reduction between these two domains.

3. Thread partitioning restores the notion of computation unit. A computation unit gathers several thread instances. For instance, a computation unit can be made of all threads directly inside a given ambient, or all threads of a same server session. Then, we count the occurrence number of threads inside each computation unit. This analysis is useful in proving high-level properties:

   - we capture a precise description of ambient contents;
   - we prove the specification correctness of a shared-memory with dynamic allocation (written in the $\pi$-calculus), by detecting that no simultaneous emission over the channels that encode memory cells may happen;
   - it proves authentication property in the case of cryptographic protocols, by detecting that for a given session identifier $A$, an $end(A)$ signal may never be launched before the corresponding $begin(A)$ signal.

## 1.2 Overview

### 1.2.1 Non standard semantics

We introduce in Chap. 2 a non standard semantics for the $\pi$-calculus. The non standard semantics is a refined semantics which aims at explicitly specifying the links between the channels and the instances of threads which have opened them.

Any instance of thread is identified unambiguously by a marker in order to distinguish that instance from all other instances. Each time a channel is opened, the name of this channel is tagged with the marker of the thread instance which has opened this channel, so that the origin of channel names is easily traced. Venet, in [74], has presented such a non standard semantics, but it applies only to a small part of the $\pi$-calculus, called *the friendly systems* [54]. In particular, it requires replication guards not to be nested, and the system to be closed. We propose here a new non standard semantics in order to relax those restrictions.

In the non standard semantics, the state of the system is described by a set of thread instances. Each thread instance is a triplet $(t,id,E)$ where $t$ denotes a program point, $id$ is the marker of the instance, and $E$ is an environment mapping variables to pairs $(x,id_x)$. Moreover, the pair $(x,id_x)$ denotes the name of the channel opened by the instance of the restriction $(\nu\ x)$ that is tagged with the marker $id_x$. While threads are running, environments are calculated in order to mimic the standard semantics.

In Chap. 3, we propose a non standard semantics for the ambient calculus. To deal with locations, we distinguish two kinds of threads: some threads denote some capabilities and some others denote ambients. Moreover, we associate each thread to a fourth component. This fourth component encodes the location of the thread: this is a pointer to an ambient thread. During interactions, locations are used to check spatial preconditions. Migrations are modeled by updating the location of the ambients. Opening an ambient consists in removing the ambient thread and in replacing in the whole system any pointer to this ambient with a pointer to its surrounding ambient.

### 1.2.2 Meta language

In Chap. 4, we propose a generic framework for describing the non standard semantics of models for mobile systems. Each encoded model is defined by some generic partial interactions and by some generic rules. Partial interactions are computed by threads. Generic rules describe the behavior of threads when they synchronize the computation of their partial interactions. Then, the syntax of a system is given by a set of program points and a static description of the partial interactions that can be performed at each program point. The state of a system is a set of thread instances which are obtained by associating a program point with a marker and an environment (thread locations — when they exist — are stored in thread environments).

Then, the operational semantics is generically derived from the meta language. Each partial interaction selects a *locus*. A locus is a part of an environment. An interaction consists in computing some partial interactions between compatible loci. In case of success, loci allow for the definition of both a local substitution and a global substitution. The local substitution updates the environment of computed threads. This essentially describes value passing between several threads: this models name passing and ambient migration (whenever local substitution is applied to thread locations). The global substitution is applied to the whole system. This substitution describes thread re-addressing: it models ambient opening (the full content of an opened ambient is re-addressed onto the surrounding ambient).

Informally, in the $\pi$-calculus, the reception $a?[x].P$ ($a$ and $x$ must be seen as variables associated to some names at run-time) is associated with the partial interaction $(in, [a], [x], \text{label}(P))$. This way we can perform an action of type *in*, with a sequence $[a]$ of parameters (i.e. the thread locus). This binds the sequence $[x]$ of variables when launching a continuation $P$. Then, a formal rule describes the communication between two threads as follows:

$$\left([1 \rightarrow in, 2 \rightarrow out], \left\{X_1^1 = X_1^2\right\}, \left[Y_1^1 \leftarrow X_2^2\right], []\right),$$

which means that the communication is enabled whenever there are two available distinct threads such that:

1. the first one — called the receiver — computes an action of type *in*;
2. the second one — called the sender — computes an action of type *out*;
3. the first arguments of the receiver and of the sender are associated to the same name at run-time.

As a result, the first bound variable of the receiver is associated with the name that is associated with the second parameter of the sender. To describe ambient opening, we also need to define a global substitution. For instance, the formal substitution $[X_1^1 \mapsto X_1^2]$ means that any instance of the value of the first parameter of the first thread is replaced with the value of the first parameter of the second thread. During an interaction, compatibility conditions, local substitution, and global substitution are computed only from the formal rule and the tuple of thread loci.

We provide some sufficient assumptions over the syntax encoding of the systems to ensure that the marker allocation scheme is not ambiguous. In Chap. 5, we use this meta language to encode the most frequently used models for mobility in the current literature. These encoding satisfy marker unambiguity sufficient assumptions. We provide an encoding of a version of the $\pi$-calculus, the *join*-calculus, the *spi*-calculus, the *ambients*, and the BIO-*ambients*. This way, our meta language allows for the description of: internal choices and external choices, guarded replication and explicit recursion, location, migration, and dissolution, term construction and term destruction, safe migration, and channeled communication across boundaries.

Our meta language may also mix these features easily. Nevertheless we cannot model the bang operator, since spontaneous replication prevents us from tracking the thread history. Moreover, we cannot deal with an equational theory such as in the applied $\pi$-calculus [1], or with symmetric communications such as in the solo-calculus [49,48] or in the fusion-calculus [65], since this feature may destroy the origin of the values that are used in the system. We have also avoided term unification (in the *spi*-calculus, we only match $k$-depth terms, where $k$ is a parameter). We have left as future works higher order communications [70] although our meta language can model them.

### 1.2.3 Context free semantics

Until now, we have only considered closed systems. However, a mobile system is usually a small part of a bigger system. The rest of the bigger system is called the context. The system part that we consider is called an open system. The open system aims at interacting with its context. Nevertheless this context may not be known: It can be made of some trusted threads or of an hostile intruder that may either force the open system into violating its usual behaviors or exploit security leaks in order to spy some sensitive information of the open system. We propose to abstract the context by the set of the values (names, locations, pointer,...) shared with the system. The semantics for open systems is then derived from the semantics for closed systems. Our context abstraction is sound for any encoded model. Moreover, we prove that it is also complete in the case of the $\pi$-calculus. However, it is complete neither in the case of the *spi*-calculus, nor in the case of mobile *ambients*, because it gives too much control to the context. With our abstraction:

1. in the case of the *spi*-calculus, the context may change dynamically the terms it has already sent to the system;
2. in the case of mobile *ambients*, the context may access to an ambient only by using its address (while ignoring the full rooting path).

We propose an encoding of this abstraction at the meta language level. This way, the analysis of an open system boils down to the analysis of a closed system (in an enriched language). We also provides generic libraries that implement sets in the meta language, so that any set-based context abstraction can be encoded at the meta language level.

### 1.2.4 Abstraction interpretation

Abstract interpretation is a theory of the approximation of semantics. It formalizes the idea that the semantics can be more or less precise according to the considered level of observation. In static analysis, abstract interpretation is used to derive a decidable semantics from a concrete — non decidable — one. Because of the upper-approximation, the result is not complete: this means that not all the properties of programs are discovered, nevertheless, the result is sound: this means that all the captured properties are satisfied in the concrete semantics.

In Chap. 7, we introduce a generic abstraction to approximate the behavior of a mobile system. It could indeed apply to any transition system. This abstraction does not depend on the abstracted properties yet: they are left as a parameter of our analysis. Hence, our framework is highly generic, and we can make a reduced product between several analyses. Comparing abstract semantics is usually a very difficult task. We provide local comparison criteria that allow for the comparison of two abstract semantics.

### 1.2.5 Environment approximation

In Chap.8, we propose an analysis that abstracts the markers and the environments of the threads. We use two granularity levels. We abstract system states by separating information about each program point — the abstraction of information about threads at a given program point is called an *abstract atom*. Then, when threads interact, we need to collect relational information about several threads. For that purpose, we use more precise abstractions — called *abstract molecules* — that describe relations among abstract atoms. Atom and molecule abstraction is fully parametric.

We propose a first class of abstraction: for each thread and each value in this thread, we abstract the relation among this value and the marker of the thread (we abstract away relations among several values in the same thread). Abstract atoms require parametric abstraction of marker pairs and abstract molecules require parametric abstraction of marker tuples. We propose ten instantiations for these abstractions and establish a hierarchy among them. This hierarchy is not only a local comparison among the abstractions, but it also ensures a comparison among the abstract semantics (i.e the analysis results). Accurate analyses rely on the use of a reduced product between a regular approximation and a relational approximation. The regular approximation abstracts the shape of each marker without any relation among them. The relational abstraction compares relations among the occurrence number of each pattern inside markers. Relational information allow for a non-uniform description of the links between threads in the sense that we distinguish between recursive instances of threads. We can prove, for instance, that a name can be communicated to at most one thread instance, and not to the other ones. In the case of the *ftp*-server, we detect that the server may return a query only to the client which has sent the corresponding request, even in the case where an unbounded number of clients are created.

This first analysis can only capture comparisons between thread markers and the values that are associated with their variables. In some cases, it turns out that there are no such relations whereas some relations between the values that are associated with several variables are useful. That is why we also propose a wider class of abstract domains. These domains can especially express some relations between values (names, locations, memory cells), even if there is no relation between their markers and the marker of their thread instance. Nevertheless, this raises some complexity problems we solve these problems by designing several domains: there is a trade-off between information partitioning, and the accuracy of information propagation. We use two complex domains. A first domain globally abstract thread properties. Information propagation is very easy, but abstract computation looses too much information. This first domain is refined by another one which partitions properties about threads according to the value labels (i.e. we abstract a set of threads by a function mapping each partition class to the set of the threads in this class). This partitioning provides very accurate properties. But constraint propagation would be too costly in such a domain. There is a balance between these domains. On the first hand, the first one partitions its own properties to refine the second one. Then, the second one applies accurate transfer functions (checks precondition satisfiability), and sends the result to the first domain. On the second hand, to perform information closure, the second domain merges all the partition cases, sends the result to the global abstraction, and partitions the result to refine its own constraints. This is a way to abstract constraint propagations in a less costly domain.

### 1.2.6 Occurrence number approximation

In Chap. 9, we propose to count the occurrence number of threads during computation sequences. It is especially useful to detect mutual exclusion. It also helps in discovering sound bounds for the number of agents during computation sequences, so that we can verify that some part of the system will not exceed the physical limits imposed by the implementation of the system. In the case of the *ftp*-server, we can automatically infer the maximum number of simultaneous client sessions.

We first abstract away any information about markers and environments. This way, a configuration is just seen as a vector that associates each program point to the number of threads at this program point. Abstract interaction is computed in two steps. First we take into account that interacting threads occur simultaneously: in the abstract, we check that some interval constraints are satisfiable and enforce them in such a case. Then, we count both the threads that are consumed and the threads that are created during the interaction: in the abstract, we apply a translation over abstract vectors. Thus, we need an abstract domain such that we can check and enforce interval constraints and such that we meet abstract elements that are closed by some translations: we use a reduced product between the domain of positive intervals and the domain of linear equalities. In such a domain, mutual exclusion and semaphores are captured by linear equalities. A mutual exclusion among some threads at program points $p_1,\ldots,p_n$ will be captured by the constraint $\sum_i x_i = 1$ where $x_i$ denotes the number of occurrence of threads at program point $p_i$. A semaphore is captured by a constraint of the form $\sum_i x_i = k$ where $k$ is the maximum number of threads that can simultaneously computed. A thread at program point $p_i$ denotes either an available token, or a concurrent agent that is using a token. We use these affine constraints to refine interval constraints about occurrence number of threads. For instance, we may have the constraints $x_1 + x_4 = 7 \wedge 0 \leq x_1 \leq 7 \wedge 0 \leq x_4 \leq 7$. We suppose that we perform an abstract transition that requires that $x_1 \geq 1$ and that computes the translation $[x_1 \leftarrow x_1 - 1; x_4 \leftarrow x_4 + 1]$. The affine invariant holds. Moreover, we can deduce the constraint $x_2 \leq 6$ from the constraints $x_1 + x_4 = 7$ and $x_1 \geq 1$. We conclude that the constraint $x_2 \leq 7$ is still valid after the abstract interaction. The affine constraint $x_1 + x_4 = 7$ is usually not explicit. It may be the result of a linear combination of several explicit affine constraints.

The affine equality domain uses Gaussian elimination to normalize equality systems. For instance, the normalized form of the system $x_1 + x_4 = 7 \wedge x_1 - x_3 = 2 \wedge x_2 - x_4 = 4$ (in which the constraint $x_1 + x_4 = 7$ is explicit) is given by the system $x_1 - x_3 = 2 \wedge x_2 - x_4 = 4 \wedge x_3 - x_4 = 5$ (in which the constraint $x_1 + x_4 = 7$ is implicit). Unfortunately, the computation of all implicit linear combinations (up to the multiplication by a constant) may lead to time blow up. Complexity issues are solved by using approximate algorithms for computing a reduction between these domains. Approximation choices mainly boil down to select which implicit constraints are to be rebuilt. Our strategy consists in solving undefined forms (i.e. a subtraction between two unbounded intervals) as much as possible.

### 1.2.7 Thread partitioning

Environment analysis focuses on the potential links between threads. This analysis captures dynamic aspects of systems, since it distinguishes among the distinct instances of syntactic objects. Unfortunately, environment analysis abstracts away concurrency, since it abstracts each thread separately. It forgets away that some threads may not occur simultaneously. On the other side, occurrence counting analysis focuses on the concurrency properties. this analysis captures the threads that can occur simultaneously. But it abstracts away the potential links between threads. A Cartesian product between these two analyses gives accurate results. It implicitly uses the coalescent product: a global interaction is enabled in the product analysis, only if it is enabled in each analysis. Unfortunately, that is the only reduction that is performed.

In Chap. 10, we propose an analysis that mixes both dynamic (it distinguishes among several instances of each computation unit) and concurrency properties. The main idea comes from the analysis of the content of mobile ambients. In mobile ambients, each ambient denotes a computation unit, so it is relevant to abstract the content of each ambient separately. We generalize this approach to models where the notion of computation unit is not explicit. Thus, we specify as a parameter what a computation unit is. We propose to partition the set of threads according to dynamical information (such as the value of a given variable). Each class is called a computation unit. Then, we count the number of thread instances in each computation unit. We succeed in proving the absence of race-conditions in a shared memory with dynamic allocation. We also prove the *non-injective agreement* property in a version of the Woo and Lam one-way public-key authentication protocol.

## 1.3 Related works

### 1.3.1 Flow analysis

Control flow analyses focus on the explicit flow of information. Nielson *et al.*use abstract interpretation in [11, 9, 10] to infer a uniform description of the interactions in a mobile system written in the $\pi$-calculus and apply Seidl's solver to get a cubic implementation of their analysis in [62]. Several versions of the framework have been proposed according to the chosen mobility model: an analysis is proposed in [59] for the *ambient* calculus ; an analysis for the *spi*-calculus is proposed in [61]; an analysis for the BIO-*ambients* is proposed in [60]. Hennessy and Riely have designed a type-based analysis of the $\pi$-calculus with the same expressive power in [44]. These analyses use explicit information flow to detect whether some security constraints specified using a security level cannot be violated. Nevertheless, these analyses are uniform (or *mono-variant*). They cannot distinguish between distinct instances of the same agent. For example, it is impossible to give distinct security levels to distinct instances of the same agent. System specification could be rewritten so that several instances of a given agent are syntacticly distinguished from the others. Therefore, this requires a human intervention to guess which replication have to be syntactically unfolded and how many instances have to be distinguished. Our analysis requires no human analysis, and can find interesting properties even if an arbitrary number of instances have to be distinguished. Moreover, it is not obvious in many cases that there exists a syntactic rewriting of the system so that several different recursive instances can be distinguished on purely syntactic ground and where the security policy is checkable using purely uniform analyses.

Cardelli *et al.* in [16, 15] use the notion of group for describing confinement properties in the $\pi$-calculus and in mobile ambients. Groups gather some values that have been declared by the same recursive instance of a thread. Then, the type system ensures that the values of one group cannot be communicated to the variables of the other groups. This ensures information confinement even inside recursive instances. However, the type system cannot validate a system where a given value first exits the scope of the thread instance which has declared it and then enters again the recursive instance which has declared it. We achieve this goal by abstracting algebraic comparisons between our history markers.

Aziz uses $k$-limiting to capture non-uniform properties in [5, 6] in the case of the $\pi$-calculus and in [5, 4] in the case of the *spi*-calculus. This approach consists in tagging each instance of a name with an integer. The analysis uses a finite name abstraction, by collapsing any name instance the index of which is greater than a fixed constant $k$. Nevertheless, this $k$-limiting abstraction has several drawbacks. If the number of simultaneously instances of a given names is not bounded (or more precisely if the analysis fails in bounding this number), the analysis may capture only uniform properties. Moreover, The choice of the parameter $k$ cannot be done without knowing the required number of name instances. Last, our markers contain much more information such as the fact that two names have been declared at the same instance of a thread, which cannot be expressed when using the $k$-limiting approach.

Venet has already proposed a non-uniform analysis in [73, 74]. This analysis infers a sound non-uniform description of the topology of communications between the agents of *friendly systems* [54], in which replication guards cannot be nested and systems are closed. The main contributions with respect to Venet's work are:

1. the extension of the non-uniform analysis to systems written in the most frequently used models (with nested replication guards);
2. the extension to open systems acting in a possibly unknown context;
3. the occurrence counting analysis;
4. the thread partitioning.

### 1.3.2 Occurrence counting analysis

Very few analyses for counting occurrences of agents have been published. Nevertheless, this problem is very close to the problem of approximating the behavior of a Petri net, and of occurrence counting in mobile ambients. In [43], Nielson *et al.* propose an exponential analysis for counting occurrences of agents inside ambients. In [63], they use context-dependent counts for inferring a more accurate description of the internal structure of agents, at the expense of a higher time complexity (an exponential number of agents are distinguished). These analyses rely on the use of a non-relational domain to abstract the content of an ambient. Then, they use disjunctive completion, and abstract any potential content of a syntactic ambient in the power set of this abstract domain. These two analyses encounter the same problem: in case several instances of the same agent may coexist, when one instance of this agent performs a computation step, these analyses cannot decide whether only one or several instances remain after this computation step, so they have to consider both cases, which leads to both a loss of precision and an exponential explosion in complexity. The use of an approximated reduced product between a relational domain and a non-relational domain to globally abstract sets of multi-sets of agents allows us to solve this problem efficiently. Thus, we obtain a very accurate analysis which is polynomial in the number of program points (i.e. polynomial in the size of the initial system configuration).

### 1.3.3 Thread partitioning

Our thread partitioning has been inspired by the work of Nielson for abstracting the content of an ambient [63]. More appropriate domains allow for designing very accurate analyses, that take benefit for the structure of the model. In mobile ambients, the content of an ambient denotes a computation unit, so it is very natural to abstract each ambient content separately.

Our contribution is to define a parametric notion of computation unit, so that we can extend the ambient content analysis to other mobility models. Our goal is to analyze separately each session of a server, or of a protocol. This way we can detect mutual exclusion inside a given session, whereas our previous occurrence counting analysis only infer mutual exclusion among program points (abstracting away both markers and environments).

### 1.3.4 Behavioral types

In [46], Kobayashi and Igarashi use CCS processes as types for mobile systems written in the $\pi$-calculus and check some behavioral invariants expressed in modal logic. Nevertheless, describing causality between actions leads to an explosion of the size of the types. Another problem is that their type system cannot express properties that deal with the dynamic creation of channel names. Rajamani and Rehof have extended this type system in [67, 18], so that it handles dynamic name creation. But type checking is undecidable in general. So, they will have to propose an approximation in future work.

### 1.3.5 Modular analysis

context independent semantics is an important issue in static analysis. It allows for analyzing only a single part of a system, without much knowledge of its context. It can be used to abstract the behavior of an instance of an agent, and to detect which names may escape the scope of this part. This can be used to detect dead code, for instance. Rajamani and Rehof propose a modular analysis in [67, 18]. Having abstracted the behavior of two modules, they can calculate an approximation of the parallel composition of them. But this analysis is very restrictive because module types must satisfy some *assume-guarantee* properties.

## 1.4 Outline

We introduce in Chap. 2 a non standard semantics for the $\pi$-calculus. In Chap. 3, we give a non standard semantics for the *ambient* calculus : this way we can deal with locations. In Chap. 4, we introduce our meta language. In Chap. 5, we encode the most frequently used models of the current literature. In Chap. 6, we explain how we can deal with open systems embedded within an unknown contexts. A generic abstract analysis is designed in Chap. 7. It is instantiated in both Chap. 8 and Chap. 9 to get, respectively, an analysis of the linkage of agents and an occurrence counting analysis. In Chap. 10, we introduce the notion of thread partitioning and provide some other analyses.

# Chapter 2

# Non standard semantics for the $\pi$-calculus

We introduce in this chapter the $\pi$-calculus and two semantics for it. The $\pi$-calculus is a formalism used to describe mobile systems. It describes a system as a set of threads which exchange information over channels. These communications enable thread synchronization, but also dynamic modification of the system topology: threads can open new channels, they can also pass control over some channels to other threads, and they can even dynamically create other threads.

We first recall a standard semantics for the $\pi$-calculus in the Sect. 2.1. We will notice that this semantics does not allow the specification of some interesting properties, because the link between the instance of threads and the names of the channels that they have opened (see also Remark 2.1.3) is not encoded explicitly. Thus we introduce in the Sect. 2.2 a refined semantics, called the non standard semantics [32, 36], where this relationship is explicitly described.

## 2.1 Standard semantics

### 2.1.1 Syntax

Here, we consider a lazy version of the synchronous polyadic $\pi$-calculus [54] with internal choice operator. In the polyadic $\pi$-calculus, threads can communicate tuples of channel names. We use the lazy version of replication introduced in [72, Chap. 7]: thread creations are performed only when they are required by a communication. This is not a limitation as full replication can be encoded with lazy replication (Cf. [72, page:102]).

Let $\mathcal{N}$ be a countable set of channel names and $\mathcal{L}$ a countable set of labels. The syntax of threads is described in Fig. 2.1(a). Syntactic components are identified by distinct labels in $\mathscr{L}$. Input guard, replication guard and name restriction act as name binders, i.e in the threads $c?^{j}[x_1,\ldots,x_n]P$, $*d?^{j}[y_1,\ldots,y_p]Q$ and $(\nu\ x)R$, the occurrences of $x_1$, …, $x_n$ in $P$, $y_1$, …, $y_p$ in $Q$ and $x$ in $R$ are bound. We also assume that no name occurs twice in a whole system, as an argument of an input guard, a replication guard or a name restriction. Usual rules about scope, substitution and $\alpha$-conversion apply. We denote by $fn(P)$ the set of free names in $P$, i.e names that are not under the scope of a binder, and by $bn(P)$ the set of bound names in $P$.

### 2.1.2 Semantics

We now informally introduce the semantics of the $\pi$-calculus. The thread $aP$ first computes the action $a$ before launching the continuation $P$. The thread $(\nu\ x)P$ opens a new channel, named $x$, the thread $P$ can use this channel for communicating, it can also send the name $x$ to the other threads. In $(P \mid Q)$, $P$ and $Q$ are two concurrent threads which may behave independently, or interact by communicating. The formula $(P \oplus Q)$ denotes an internal choice between two threads. Either $P$ or $Q$ is run, while the other fades away; the choice between $P$ and $Q$ does not depend on the other threads. The thread $\mathbf{0}$ does nothing. The thread $c!^{i}[x_1,\ldots,x_n]P$ sends a message via the channel named $c$, this message is the tuple of channel names $(x_1,\ldots,x_n)$. The thread $c?^{i}[y_1,\ldots,y_n]P$ waits for a message on the channel named $c$, and binds the channel names $y_1$,…,$y_n$ to the received channel names. The thread $*c?^{i}[y_1,\ldots,y_n]P$ is a *resource*: it replicates itself just before receiving a message: a new instance of $P$ is launched with $y_1$,…,$y_n$ bound to the received channel names while $*c?^{i}[y_1,\ldots,y_n]P$ waits for the next message.

The operational semantics is given by both a congruence relation in Fig. 2.1(b) and a reduction relation in Fig. 2.1(c). The congruence relation allows threads to interact, while the reduction relation describes thread computations. Some rules in the congruence relation let threads move inside the syntactic tree: they assert the associativity and commutativity of the parallel composition. Some others extend the scope of names to the threads they are communicated to: $\alpha$-conversion solves conflicts between names, swapping selects the name the scope of which we wish to extend, and extrusion extends its scope to another thread. The reduction relation describes threads' communications. A communication is allowed when there are two concurrent threads, such that the first one sends a message on a channel, while the second one waits for a message on the same channel (we also request that both messages have the same arity). The results of such a communication are obtained by applying the substitution of the $\lambda$-calculus in the continuation of the message receiver. When the receiver is a resource, it is just syntactically replicated before performing the communication; this way the resource is still available after the communication. We have labeled each choice reduction step with the symbol $\oplus$

$$\begin{array}{lcll}
P & ::= & aP & \text{(action)} \\
 & | & (\nu\, x)P & \text{(name restriction)} \\
 & | & (P \mid P) & \text{(parallel composition)} \\
 & | & (P \oplus P) & \text{(internal choice)} \\
 & | & \mathbf{0} & \text{(nil)} \\
a & ::= & c!^{j}[x_1,\dots,x_n] & \text{(output guard )} \\
 & | & c?^{i}[x_1,\dots,x_n] & \text{(input guard)} \\
 & | & *c?^{i}[x_1,\dots,x_n] & \text{(replication guard)}
\end{array}$$

where $c, x_1, \dots, x_n, x \in \mathscr{N}$, $i, j \in \mathscr{L}$ *and* $n \geqslant 0$.

(a) Syntax.

$$\begin{array}{rcll}
(\nu\, x)P & \equiv & (\nu\, y)P[x \leftarrow y] \quad \text{if } y \notin fn(P) & (\alpha\text{-conversion}) \\
P \mid Q & \equiv & Q \mid P & \text{(commutativity)} \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & \text{(associativity)} \\
P \mid \mathbf{0} & \equiv & P & \text{(end of a thread)} \\
(\nu\, x)\mathbf{0} & \equiv & \mathbf{0} & \text{(garbage collecting)} \\
(\nu\, x)(\nu\, y)P & \equiv & (\nu\, y)(\nu\, x)P & \text{(swapping)} \\
((\nu\, x)P) \mid Q & \equiv & (\nu\, x)(P \mid Q) \quad \text{if } x \notin fn(Q) & \text{(extrusion)}
\end{array}$$

where $x, y \in \mathscr{N}$.

(b) Congruence relation.

$$\begin{array}{rl}
c!^{j}[x_1,\dots,x_n]P \mid c?^{i}[y_1,\dots,y_n]Q \xrightarrow{(i,j)} P \mid \widetilde{Q} & \text{(comm.)} \\
c!^{j}[x_1,\dots,x_n]P \mid *c?^{i}[y_1,\dots,y_n]Q \xrightarrow{(i,j)} P \mid \widetilde{Q} \mid *c?^{i}[y_1,\dots,y_n]Q & \text{(replication)} \\
P \oplus Q \xrightarrow{\oplus} P & \text{(left choice)} \\
P \oplus Q \xrightarrow{\oplus} Q & \text{(right choice)}
\end{array}$$

$$\frac{P \xrightarrow{\lambda} Q}{(\nu\, x)P \xrightarrow{\lambda} (\nu\, x)Q} \qquad \frac{P' \equiv P \quad P \xrightarrow{\lambda} Q \quad Q \equiv Q'}{P' \xrightarrow{\lambda} Q'} \qquad \frac{P \xrightarrow{\lambda} P'}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

where $c, x, x_1, \dots, x_n, y_1, \dots, y_n \in \mathscr{N}$, $i, j \in \mathscr{L}$, $\lambda \in \{\oplus\} \cup (\mathscr{L} \times \mathscr{L})$, and $\widetilde{Q} = Q[y_1 \leftarrow x_1, \dots, y_n \leftarrow x_n]$.

(c) Reduction relation.

Figure 2.1: Standard operational semantics.

and each communication reduction step with the labels of both threads involved in the communication: this will allow us to relate the state of a system to the history of the computation steps that have led to this state.

### 2.1.3 Examples

We now propose some examples to illustrate both this semantics and the kind of properties we are interested in. We will find that the semantics we have considered is not precise enough to handle the properties we are interested in.

**Example 2.1.1.** *An* ftp-*server can be described by the system given in Fig. 2.2.*

*The first resource repeatedly creates a new client which sends a query to a server. This query is composed of a request* ***request****, and an address* ***address****. The client sends its query again in the case that it receives a failure report denoted by the thread* ***address![]****. The second resource describes the server. When this one receives a query, it replicates itself. Then, either it uses an available port and computes the query or it reports a failure to the client by spawning the thread* ***address![]****. Available ports are denoted by threads* ***port![]****. Data processing just consists in a communication between two threads of the server, through the channel the name of which is* ***deal****: most computational features are abstracted away. After this communication, the port is released, while the answer is sent back to the client. An instance of the thread* ***email***![***rep***] *is left as a trace of the session.*

*Our analysis will prove both the integrity and the non-exhaustion of this system: it will discover that each time a thread* ***email![rep]*** *is spawned, the names* ***email*** *and* ***rep*** *are respectively bound to the names of two channels opened by the restrictions* ($\nu$ ***address***) *and* ($\nu$ ***request***) *of the same instance of a resource (see Chap. 8), and thus the server returns its computed answer to the correct client; it also captures the fact that no more than three instances of the syntactic thread* ***deal***![***data***] *can occur simultaneously (see Chap. 9), which means that no more than three simultaneous sessions can be active in the same time.*

**Example 2.1.2.** *We propose in Fig. 2.3 a mobile system which creates a ring of processes, with a token passed around this ring. The names of the channels opened by name restrictions* ($\nu$ **left0**) *and* ($\nu$ ***right***) *denote the processes of the ring. The first part of the system describes the ring creation. The first process is created by the restriction* ($\nu$ **left0**)*. A thread* **mon**![***v1***;***v2***] *denotes a connection between two processes. Then, each time the first resource is replicated, a new process is created and linked to the previous process, which was passed as an argument of the replication. The second resource replication closes the ring by linking the last created process to the first created process. The second part of the system describes the execution of the processes: an additional resource spawns a resource for each process of the ring. Then a token is put into the ring of processes: the*

```
$((\nu\ \mathrm{make})(\nu\ \mathrm{server})(\nu\ \mathrm{port})$
$((*\mathrm{make}?^{1}[\,](\nu\ address)(\nu\ request)$
           $($
            $(*address?^{2}[\,]\mathrm{server}!^{3}[address, request])$
           $|$
            $address!^{4}[\,]$
           $|$
            $\mathrm{make}!^{5}[\,]$
           $))$
$|$
 $(*\mathrm{server}?^{6}[email, data](\nu\ deal)$
           $($
            $\mathrm{port}?^{7}[\,](deal!^{8}[data] \mid deal?^{9}[rep](email!^{10}[rep] \mid \mathrm{port}!^{11}[\,]))$
           $\oplus$
            $email!^{12}[\,]$
           $))$
$\mid \mathrm{port}!^{13}[\,]$
$\mid \mathrm{port}!^{14}[\,]$
$\mid \mathrm{port}!^{15}[\,]$
$\mid \mathrm{make}!^{16}[\,])$
$)$
```

Figure 2.2: An *ftp*-server.

```
$((\nu\ \mathrm{make})(\nu\ \mathrm{mon})(\nu\ \mathrm{left0})$
$($
 $(\ (*\mathrm{make}?^{1}[left](\nu\ right)(\mathrm{mon}!^{2}[left, right] \mid \mathrm{make}!^{3}[right]))$
 $\mid (*\mathrm{make}?^{4}[left](\mathrm{mon}!^{5}[left, \mathrm{left0}]))$
 $\mid \mathrm{make}!^{6}[\mathrm{left0}])$
$|$
 $((*\mathrm{mon}?^{7}[prev, next]$
           $(*prev?^{8}[\,](\nu\ crit)(crit?[\,]^{9}next!^{10}[\,]$
                                 $\mid crit!^{11}[\,])))$
 $\mid \mathrm{left0}!^{12}[\,]))$
$)$
```

Figure 2.3: A ring of processes.

*token is denoted by syntactic copies of the threads* $\textit{\textbf{next}}![]$ *and* $\textbf{left0}![]$. *The name of this thread describes the token location. When the token is available, the corresponding process can replicate its resource, and as a result the process enters its critical section. The critical section is exited when the two threads* $\textit{\textbf{crit}}![]$ *and* $\textit{\textbf{crit}}?[]$ *have interacted; the token is then passed to the next process.*

*Our analysis can prove both the integrity and the non-exhaustion of this system: it discovers that each time a thread* $\textbf{mon}![\textit{\textbf{left}};\textit{\textbf{right}}]$ *is spawned, either the name **left** is linked to the channel opened by the restriction* $(\nu\ \textbf{left0})$, *or both names **left** and **right** are linked to two channels opened by instances of the restriction* $(\nu\ \textit{\textbf{right}})$, *but the channel linked to the name **left** has been opened by the previous instance of it, which means that a process of the ring can only be connected to either the first one or to the next one (see Chap. 8); it captures the fact that only one simultaneous instance of the syntactic thread* $\textit{\textbf{crit}}![]$ *can exist (see Chap. 9). That is to say, that only one process of the ring can enter its critical section at a given time.*

**Remark 2.1.3.** *The standard semantics is not well suited to express and capture integrity properties, because the link between thread instances and the names of the channels they have opened is not encoded explicitly. For instance, if we think about the example of the* ftp-*server and if we cleverly choose the names of opened channels by indexing them with the instance number of the client resource, we obtain after two sessions of the server a system of the following form*[1]*:*

$$\begin{array}{l}(\nu\ \overline{c})(\nu\ \mathit{address}_1)(\nu\ \mathit{request}_1)(\nu\ \mathit{address}_2)(\nu\ \mathit{request}_2)\\ \quad(\mathscr{S}' \mid \mathit{address}_1!^{10}[\mathit{request}_1] \mid \mathit{address}_2!^{10}[\mathit{request}_2]).\end{array}$$

*It appears explicitly that request answers are returned at good addresses. However, we could have chosen the names differently and obtained the following* $\alpha$-*equivalent configuration:*

$$\begin{array}{l}(\nu\ \overline{c})(\nu\ \mathit{address}_2)(\nu\ \mathit{request}_1)(\nu\ \mathit{address}_1)(\nu\ \mathit{request}_2)\\ \quad(\mathscr{S}' \mid \mathit{address}_2!^{10}[\mathit{request}_1] \mid \mathit{address}_1!^{10}[\mathit{request}_2])\end{array}$$

*in which this property is lost.*

*The link between the recursive instances of a thread and the names of the channels they have opened could be easily hard-coded: it would be enough to open a new channel named* **p** *for each recursive instance of a thread, and then encoding the relation that this instance has opened a given channel name* **n** *by*

[1]This term only shows explicitly the information we are interested in. The variable $\mathscr{S}'$ denotes the rest of the system and the notation $(\nu\ \overline{c})$ denotes a sequence of restrictions for all implicit names.

*spawning a thread* $\mathbf{has\_opened}!^{\mathbf{i}}[\mathbf{p};\mathbf{n}]$*, where* **has_opened** *is the name of a channel opened at the beginning of the system computation. Nevertheless, it would be very difficult to abstract this relation. All the more so since we are also interested in more complex properties, such as whether two channels have to be opened by two successive instances of a thread. Moreover, we do not know statically which complex properties are required to prove easier ones.*

The purpose of the next section is to design a semantics in which channel name origin is carefully traced and can easily be abstracted.

## 2.2 Refined semantics

The non standard semantics is a refined semantics which aims at explicitly specifying the links between the channels and the instances of threads which have opened them. Any instance of a thread is identified unambiguously by a marker in order to distinguish that instance from all others. Each time a channel is opened, the name of this channel is tagged with the marker of the thread instance which has opened this channel, so that the origin of channel names is easily traced. Venet, in [74], has presented such a non standard semantics, but it applies only to a small part of the $\pi$-calculus, called *the friendly systems* [54]. In particular, it requires replication guards not to be nested, and the system to be closed. We propose here a new non standard semantics in order to relax those restrictions.

This section will be organized as follows: we first describe our marker allocation scheme, then we propose a naive fully operational semantics for describing the behavior of closed mobile systems, and we finally improve this semantics in order to reduce the number of computation steps.

### 2.2.1 Fresh name allocation

As explained before, $\alpha$-conversion prevents us from expressing the link between recursive instances and the names of the channels they have opened. To avoid the use of $\alpha$-conversion, we propose a name allocation scheme which ensures the freshness of allocated names. Such a scheme has already been proposed by De Bruijn in [29]. Nevertheless, our requirement is quite different. De Bruijn's naming scheme allows $\alpha$-conversion to be avoided, in order to simplify some manual proofs. We also expect our scheme to allow us to express some integrity properties. For instance, we would like to express in our semantics the fact that two names are denoting channels which have been opened by the same instance of a given thread. Furthermore, as we want to make static analyses, we want to capture invariants on allocated markers. For that purpose, we want the scheme not to depend on the interleaving order. To solve that problem, we propose to tag each instance of thread by a marker which encodes the history of the replications which have led to its creation. Each name will then be tagged with the marker of the thread which has opened the channel that is denoted by this name.

We denote by $\mathscr{M}$ the set of all binary trees the leaves of which are all labeled with $\varepsilon$ and the nodes of which are labeled with pairs $(i,j)$ where both $i$ and $j$ are in $\mathscr{L}$. The tree having a node labeled with $a$, a left sibling $t_1$ and a right one $t_2$ is denoted by $N(a,t_1,t_2)$. Markers are binary trees in $\mathscr{M}$. Initial thread instances are tagged with the marker $\varepsilon$, while the marker of each new thread instance is calculated recursively from the marker of the thread instances the computation of which has lead to its creation:

- when a computation step does not involve replicating a resource, the marker of the computed thread is just passed to its continuation;
- when a resource is replicated, a new marker is deterministically allocated to the spawned instance: it is given by $N((i,j),id_i,id_j)$ where $i$ is the label of the resource, $id_i$ is the marker of the resource, $j$ is the label of the thread instance which replicates the resource and $id_j$ is the marker of this thread instance.

Marker allocation consistency is expressed by the following proposition:

**Proposition 2.2.1.** *During each computation sequence, two distinct instances of the same thread are always tagged with distinct markers.*

*Proof.* The proof of Prop. 2.2.1 can be made by induction on the length of the computation sequence. It relies on the fact that each tagged thread instance contains explicitly both the label and the markers of a thread instance which has necessarily been consumed to spawn this instance. $\square$

Moreover, according to the following proposition, we can simplify the shape[2] of the markers without losing marker allocation consistency:

**Proposition 2.2.2.** *Let $\phi_1$ and $\phi_2$ be the two following functions:*

$$\phi_1 : \left\{ \begin{array}{ccc} \mathscr{M} & \to & (\mathscr{L}^2)^* \\ N(a,b,c) & \mapsto & \phi_1(c).a \\ \varepsilon & \mapsto & \varepsilon \end{array} \right. \qquad \phi_2 : \left\{ \begin{array}{ccc} \mathscr{M} & \to & \mathscr{L}^* \\ N((i,j),b,c) & \mapsto & \phi_2(c).j \\ \varepsilon & \mapsto & \varepsilon. \end{array} \right.$$

*Marker allocation remains consistent when replacing each marker by its image by $\phi_1$ or $\phi_2$.*

[2]We have reversed markers in order to restore chronological order (so that the past is on the left and the future in on the right)

Such simplifications allow us to reduce the cost of our analysis, but also lead to a loss of accuracy, since they merge information related to distinct computation sequences of the system.

**Example 2.2.3.** *Coming back to the example of the* ftp*-server, with this allocation scheme, the first instance of the client resource will be tagged with the marker* $id_1 = N((1,16),\varepsilon,\varepsilon)$*, while the second instance will be tagged with the marker* $id_2 = N((1,5),\varepsilon,N((1,16),\varepsilon,\varepsilon))$*. So that the configuration reached after two sessions of the server will be of the following form:*

$$\begin{array}{l}(\nu\ \bar{\mathbf{c}})(\nu\ address_{id_1})(\nu\ request_{id_1})(\nu\ address_{id_2})(\nu\ request_{id_2})\\ \quad(\mathscr{S}' \mid address_{id_1}!^{10}[request_{id_1}] \mid address_{id_2}!^{10}[request_{id_2}]),\end{array}$$

*where the names are indexed by the marker of the threads which have opened the channels they denote. We did not indicate the marker of thread instances which depends on the number of attempts required to establish the connection with the server. It appears explicitly that the names* $\boldsymbol{address_{id_i}}$ *and* $\boldsymbol{request_{id_i}}$ *communicated to an instance of the thread labeled* **10** *denote two channels opened by the same recursive instance of a thread.*

### 2.2.2 Naive semantics

We now propose a fully operational semantics of the $\pi$-calculus, in which the channel names are allocated according to the previously proposed fresh name allocation scheme. Furthermore, we get rid of the congruence relation by orienting it, and simulating it by additional operational rules.

#### 2.2.2.1 Definition

Let us consider the case of a closed mobile system $\mathscr{S}$ in the $\pi$-calculus. The subset of $\mathscr{L}$ used in labeling $\mathscr{S}$ is denoted by $\mathscr{L}_{\text{used}}$. A *non standard configuration* is a set of thread instances, where a *thread instance* is a triplet composed of a syntactic component, a marker and an environment. The *syntactic component* is a copy of a sub-term of $\mathscr{S}$, the *marker* is calculated at the creation of the thread and the *environment* specifies the semantic value of each free name of the syntactic component: it maps each free name of the syntactic component to a pair $(x,id)$, where $x$ is a bound name of $\mathscr{S}$ and $id$ is a marker. Intuitively, $(x,id)$ refers to the name of the channel opened by the instance of the restriction $(\nu\ x)$ tagged with the marker $id$. While threads are running, environments are calculated in order to mimic the standard semantics. The translation of a labeled system $\mathscr{S}$ into a set of initial threads and non standard computation rules are given in Fig. 2.4.

$$\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) = (\mathscr{S}, \varepsilon, \emptyset)$$

(a) Initial confi guration

$$\begin{array}{rcl}
C \cup \{(P \mid Q, id, E)\} & \xrightarrow{\varepsilon}_{\mathrm{n}} & C \cup \{(P, id, E_{|fn(P)}); (Q, id, E_{|fn(Q)})\} \\
C \cup \{((\nu\, x)P, id, E)\} & \xrightarrow{\varepsilon}_{\mathrm{n}} & C \cup \{(P, id, E[x \to (x, id)]_{|fn(P)})\} \\
C \cup \{(\mathbf{0}, id, E)\} & \xrightarrow{\varepsilon}_{\mathrm{n}} & C
\end{array}$$

(b) Structural rules

$$\begin{array}{rcl}
C \cup \{P \oplus Q, id, E\} & \xrightarrow{\oplus}_{\mathrm{n}} & C \cup \{(P, id, E_{|fn(P)})\} \\
C \cup \{P \oplus Q, id, E\} & \xrightarrow{\oplus}_{\mathrm{n}} & C \cup \{(Q, id, E_{|fn(Q)})\}
\end{array}$$

(c) Choice rules

$$\frac{E_?(y) = E_!(x)}{C \cup \left\{ \begin{array}{l} (y?^i[\overline{y}]P, id_?, E_?); \\ (x!^j[\overline{x}]Q, id_!, E_!) \end{array} \right\} \xrightarrow{(i,j)}_{\mathrm{n}} C \cup \left\{ \begin{array}{l} (P, id_?, E_?[\overline{y} \to E_!(\overline{x})]_{|fn(P)}); \\ (Q, id_!, E_{!|fn(Q)}) \end{array} \right\}}$$

$$\frac{E_?(y) = E_!(x),\ id_* = N((i,j), id_?, id_!)}{C \cup \left\{ \begin{array}{l} (*y?^i[\overline{y}]P, id_?, E_?); \\ (x!^j[\overline{x}]Q, id_!, E_!) \end{array} \right\} \xrightarrow{(i,j)}_{\mathrm{n}} C \cup \left\{ \begin{array}{l} (*y?^i[\overline{y}]P, id_?, E_?); \\ (P, id_*, E_?[\overline{y} \to E_!(\overline{x})]_{|fn(P)}); \\ (Q, id_!, E_{!|fn(Q)}) \end{array} \right\}}$$

(d) Communication rules

Figure 2.4: Naive semantics.

Roughly speaking, the initial configuration contains only one thread: the system itself, tagged with the marker of the initial thread, $\varepsilon$. Since the system is closed, the environment is empty. Structural rules mimic and orient the congruence relation. A thread the syntactic component of which is composed of two concurrent threads can be replaced by the two corresponding threads. Name restriction consists in opening a channel denoted by a fresh name, and binding the corresponding variable to this name in the environment of the continuation. The fresh name is obtained by tagging the name used in the restriction by the marker of the thread which has opened the corresponding channel. A thread the syntactic component of which is the empty thread can be removed. Choice rules mimic choice reduction rules. A thread the syntactic component of which is a choice between two threads can be replaced by a thread corresponding to one of these threads. Communication rules mimic communication reduction rules. The synchronization condition is checked in the environment of the communicating threads. Name passing is described by explicit substitution in environments. In the case that a resource is replicated, a fresh marker is inferred according to our marker allocation scheme.

**Example 2.2.4.** *We consider the following system:*

$$(\nu\ \mathrm{a})(*\mathrm{a}?^1[]((\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0)) \mid \mathrm{a}!^4[]0),$$

*and propose a computation sequence for it in the naive non standard semantics in Fig. 2.5. The initial state $C_0$ is just a single thread the syntactic component of which is the system, the marker of which is $\varepsilon$ and the environment of which is empty. The first computation step $C_0 \xrightarrow{\varepsilon}_{\mathrm{n}} C_1$ consists in opening a new channel, named $(a,\varepsilon)$, since it is opened using the restriction $(\nu\ a)$ of a thread the marker of which is $\varepsilon$. The second computation step $C_1 \xrightarrow{\varepsilon}_{\mathrm{n}} C_2$ consists in decomposing the thread into two concurrent threads; thus the marker of the single thread is just passed to both threads. The third computation step $C_2 \xrightarrow{(1,4)}_{\mathrm{n}} C_3$ is a communication between the two threads. Since the first one is a resource, it is still available after the communication. The thread which has sent the message is consumed and new threads, corresponding to the continuation of the communicating threads are spawned. The continuation of the resource is tagged with a new marker $N((1,4),\varepsilon,\varepsilon)$ obtained from both the labels and the markers of both communicating threads, while the marker of the thread which has sent the message is just passed to its continuation. The next computation step, $C_3 \xrightarrow{\varepsilon}_{\mathrm{n}} C_4$, is a garbage collection: it consists in removing the thread corresponding to the empty thread. The fifth computation step $C_4 \xrightarrow{\varepsilon}_{\mathrm{n}} C_5$ opens a channel. Its name is given by $(b,N((1,4),\varepsilon,\varepsilon))$ since it is opened by the restriction $(\nu\ b)$ of a thread the marker of which is $N((1,4),\varepsilon,\varepsilon)$. Then in the computation step $C_5 \xrightarrow{\varepsilon}_{\mathrm{n}} C_6$ a*

$$C_0 \xrightarrow{\varepsilon}_{\mathrm{n}} C_1 \xrightarrow{\varepsilon}_{\mathrm{n}} C_2 \xrightarrow{(1,4)}_{\mathrm{n}} C_3 \xrightarrow{\varepsilon}_{\mathrm{n}} C_4 \xrightarrow{\varepsilon}_{\mathrm{n}} C_5 \xrightarrow{\varepsilon}_{\mathrm{n}} C_6 \xrightarrow{(1,3)}_{\mathrm{n}} C_7 \longrightarrow_{\mathrm{n}}^{*} C_8$$

where

$$C_0 = \left\{ \big((\nu\ \mathrm{a})((*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0)) \mid \mathrm{a}!^4[]0), \varepsilon, \emptyset\big) \right\}$$

$$C_1 = \left\{ \big((*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0)) \mid \mathrm{a}!^4[]0, \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big) \right\}$$

$$C_2 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big(\mathrm{a}!^4[]0, \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big) \end{array} \right\}$$

$$C_3 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big((\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), N((1,4), \varepsilon, \varepsilon), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ (0, \varepsilon, \emptyset) \end{array} \right\}$$

$$C_4 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big((\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), N((1,4), \varepsilon, \varepsilon), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big) \end{array} \right\}$$

$$C_5 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big((b!^2[b]0 \mid \mathrm{a}!^3[]0), N((1,4), \varepsilon, \varepsilon), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon), b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]\big) \end{array} \right\}$$

$$C_6 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big(b!^2[b]0, N((1,4), \varepsilon, \varepsilon), [b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]\big)\,; \\ \big(\mathrm{a}!^3[]0, N((1,4), \varepsilon, \varepsilon), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big) \end{array} \right\}$$

$$C_7 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big(b!^2[b]0, N((1,4), \varepsilon, \varepsilon), [b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]\big)\,; \\ \big((\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ (0, \varepsilon, \emptyset) \end{array} \right\}$$

$$C_8 = \left\{ \begin{array}{l} \big(*\mathrm{a}?^1[](\nu\ b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big)\,; \\ \big(b!^2[b]0, N((1,4), \varepsilon, \varepsilon), [b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]\big)\,; \\ \big(b!^2[b]0, N((1,3), \varepsilon, N((1,4), \varepsilon\varepsilon)), [b \mapsto (b, N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)))]\big)\,; \\ \big(\mathrm{a}!^3[]0, N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]\big) \end{array} \right\}$$

Figure 2.5: A computation sequence in the naive semantics.

*single thread is cut into two concurrent threads, which allows us to go on with the recursion: the computation step $C_6 \xrightarrow{(1,3)}_{\mathrm{n}} C_7$ allows the system to spawn another instance of the resource with the fresh marker $N((1,3),\varepsilon,N((1,4),\varepsilon,\varepsilon))$. There is no confusion between recursive instances and the names of the channels they have opened: we can notice that in each thread corresponding to the thread labeled $b!^2[b]$, the marker of the thread and the marker of the name communicated to the variable b are the same.* □

#### 2.2.2.2 Correspondence

The correspondence between the standard and the naive non standard semantics is established by a translation function $\Pi$. We define the translation $\Pi(C)$ of a non standard configuration $C$ as follows:

$$\Pi(C) = (\nu\, c_1)\ldots(\nu\, c_k)(\overline{E}(t_1)|\ \ldots\ |\overline{E}(t_l))$$

where $\{c_i \mid i \in [\![1;k]\!]\} = \{E(x) \mid (P,id,E) \in C,\ x \in fn(P)\}$ is the set of the names used by the system and $C = \{t_i \mid i \in [\![1;l]\!]\}$ is the set of threads and $\overline{E}$ is a function which maps each thread $(P,id,E)$ into the thread obtained by substituting each free name $x$ by its image $E(x)$ in the environment of the syntactic components $P$. The system $\Pi(C)$ is well defined thanks to associativity, commutativity, and swapping rules. We have also assumed[3] that $\mathscr{N} \times \mathscr{M}$ was a subset of $\mathscr{N}$.

The standard and the non standard semantics are in *weak bisimulation*, as expressed by the following theorem:

**Theorem 2.2.5.** *We have $\mathscr{S} = \Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}))$, and for any non standard configurations C and for any word $u \in (\mathscr{L}^2 \cup \{\varepsilon;\oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) \xrightarrow{u}{}_{\mathrm{n}}^{*} C$, we have:*

1. $C \xrightarrow{\varepsilon}_{\mathrm{n}} C' \implies \Pi(C) \equiv \Pi(C')$;

2. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\},\ C \xrightarrow{\lambda}_{\mathrm{n}} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;

3. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\},\ \Pi(C) \xrightarrow{\lambda} P \implies \exists D,\ \exists E,\ \begin{cases} C \xrightarrow{\varepsilon}{}_{\mathrm{n}}^{*} D \xrightarrow{\lambda}_{\mathrm{n}} E \\ \Pi(E) \equiv P. \end{cases}$

The proof of Thm. 2.2.5 is shown in appendix A.1.

We now propose to reduce the number of reduction steps in order to make analysis design easier, and also to obtain a more efficient analysis. We will first consider in Sect. 2.2.3 a semantics in which all structural steps are automatically performed. Then we will introduce in Sect. 2.2.4 another semantics in where choice steps are also dealt in the same way.

[3]We consider in fact that a tagged name is a name, since there exists a bijection between the set of the tagged names and the set of the names.

### 2.2.3 Strongly bisimilar semantics

We propose a semantics in which all structural steps are automatically performed:

#### 2.2.3.1 Definition

The binary relation $\xrightarrow{\varepsilon}_{\mathrm{n}}$ is nœtherian and locally confluent. So, it is confluent [30], and we can define its limit $\Downarrow$ as follows:

$$a \Downarrow b \textit{ if and only if } a \xrightarrow{\varepsilon}_{\mathrm{n}}^{*} b \textit{ and } \nexists c, b \xrightarrow{\varepsilon}_{\mathrm{n}} c.$$

We now express explicitly this limit by designing an extraction function $\beta^{\mathrm{i}}$ which performs all structural rules in parallel. The definition of $\beta^{\mathrm{i}}$ is given in Fig. 2.6.

$$\begin{aligned}
\beta^{\mathrm{i}}((\nu\, n)P, id, E) &= \beta^{\mathrm{i}}(P, id, (E[n \mapsto (n, id)])) \\
\beta^{\mathrm{i}}(\mathbf{0}, id, E) &= \emptyset \\
\beta^{\mathrm{i}}(P \mid Q, id, E) &= \beta^{\mathrm{i}}(P, id, E) \cup \beta^{\mathrm{i}}(Q, id, E) \\
\beta^{\mathrm{i}}(P \oplus Q, id, E) &= \{(P \oplus Q, id, E_{|fn(P \oplus Q)})\} \\
\beta^{\mathrm{i}}(aP, id, E) &= \{(aP, id, E_{|fn(aP)})\}
\end{aligned}$$

Figure 2.6: Extraction function.

**Proposition 2.2.6.** *For any non standard configuration $C$, we have $C \Downarrow \bigcup_{t \in C} \beta^{\mathrm{i}}(t)$.*

The proof of Prop. 2.2.6 is shown in the appendix A.2.

Then, we can define the intermediate semantics as the transition system $(\{\mathscr{C}_0^{\mathrm{i}}(\mathscr{S})\}, \longrightarrow_{\mathrm{i}})$, where the set of the initial states $\{\mathscr{C}_0^{\mathrm{i}}(\mathscr{S})\}$ and the computation rule $\longrightarrow_{\mathrm{i}}$ are given as follows:

- $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) = \beta^{\mathrm{i}}(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}))$,
- $\forall \lambda \in (\mathscr{L}^2 \cup \{\oplus\}), a \xrightarrow{\lambda}_{\mathrm{i}} b$ *if and only if* $\exists c,\ a \xrightarrow{\lambda}_{\mathrm{n}} c$ *and* $c \Downarrow b$.

**Example 2.2.7.** *We come back to the previously given system:*

$$(\nu\ \mathrm{a})(*\mathrm{a}?^1[]((\nu\, b)(b!^2[b]0 \mid \mathrm{a}!^3[]0)) \mid \mathrm{a}!^4[]0),$$

$$D_0 \xrightarrow{(1,4)}_{\mathrm{i}} D_1 \xrightarrow{(1,3)}_{\mathrm{i}} D_2$$

where

$$D_0 = \left\{ \begin{array}{l} (*\mathrm{a}?^1[](\nu\, b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]); \\ (\mathrm{a}!^4[]0, \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]) \end{array} \right\}$$

$$D_1 = \left\{ \begin{array}{l} (*\mathrm{a}?^1[](\nu\, b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]); \\ (b!^2[b]0, N((1,4), \varepsilon, \varepsilon), [b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]); \\ (\mathrm{a}!^3[]0, N((1,4), \varepsilon, \varepsilon), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]) \end{array} \right\}$$

$$D_2 = \left\{ \begin{array}{l} (*\mathrm{a}?^1[](\nu\, b)(b!^2[b]0 \mid \mathrm{a}!^3[]0), \varepsilon, [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]); \\ (b!^2[b]0, N((1,4), \varepsilon, \varepsilon), [b \mapsto (b, N((1,4), \varepsilon, \varepsilon))]); \\ (b!^2[b]0, N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)), [b \mapsto (b, N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)))]); \\ (\mathrm{a}!^3[]0, N((1,3), \varepsilon, N((1,4), \varepsilon, \varepsilon)), [\mathrm{a} \mapsto (\mathrm{a}, \varepsilon)]) \end{array} \right\}$$

Figure 2.7: A computation sequence in the intermediate semantics.

*and give a computation sequence for it in Fig. 2.7. It appears explicitly that structural computation steps are not described anymore. They are all performed implicitly at the beginning of the system, and after each communication or choice computation step: this way, the initial state $D_0$ is equal to the state $C_2$ while the whole computation sequence $C_0 \xrightarrow{\varepsilon}_{\mathrm{n}} C_1 \xrightarrow{\varepsilon}_{\mathrm{n}} C_2$ becomes implicit, then the single computation step $D_1 \xrightarrow{(1,4)}_{\mathrm{i}} D_2$ encodes the computation sequence $C_2 \xrightarrow{(1,4)}_{\mathrm{n}} C_3 \xrightarrow{\varepsilon}_{\mathrm{n}} C_4 \xrightarrow{\varepsilon}_{\mathrm{n}} C_5 \xrightarrow{\varepsilon}_{\mathrm{n}} C_6$, and the single computation step $D_2 \xrightarrow{(1,3)}_{\mathrm{i}} D_3$ encodes the computation sequence $C_6 \xrightarrow{(1,3)}_{\mathrm{n}} C_7 \longrightarrow_{\mathrm{n}}^{*} C_8$.*

#### 2.2.3.2 Correspondence

The standard and the intermediate semantics are in *strong bisimilation* (up to $\equiv$), as expressed by the following theorem:

**Theorem 2.2.8.** *We have $\mathscr{S} \equiv \Pi(\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}))$, and for all non standard configurations $C$ and for all word $u \in (\mathscr{L}^2 \cup \{\varepsilon; \oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) \xrightarrow{u}_{\mathrm{i}}^{*} C$, we have:*

1. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\}$, $C \xrightarrow{\lambda}_{\mathrm{i}} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;

2. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\}$, $\Pi(C) \xrightarrow{\lambda} P \implies \exists D, \begin{cases} C \xrightarrow{\lambda}_{\mathrm{i}} D \\ \Pi(D) \equiv P. \end{cases}$

The proof of Thm. 2.2.8 is shown in appendix A.2.

## 2.2.4 Efficient semantics

We propose to focus on communication rules and also to factor choice rules. To do this, we have to restrict the set of the traces: to make things easier, we propose to consider only the traces in which communications are delayed until no choices can be made.[4]

### 2.2.4.1 Definition

We denote by $\dashrightarrow$ the binary relation $\xrightarrow{\varepsilon}_{\mathrm{n}} \cup \xrightarrow{\oplus}_{\mathrm{n}}$. The reduction relation $\dashrightarrow$ is nœtherian, but not necessarily locally confluent. We define the relation $\Longrightarrow$ as follows:

$$a \Longrightarrow b \textit{ if and only if } a \dashrightarrow^{*} b \textit{ and } \nexists c,\ b \dashrightarrow c.$$

Since $\dashrightarrow$ is not necessarily confluent, $\Longrightarrow$ may be not deterministic. We now express explicitly its action by designing an extraction function $\beta$ which computes the set of all the successors of a given configuration. The definition of $\beta$ is given in Fig. 2.8(a).

**Proposition 2.2.9.** *For any non standard configuration $C$, we have:*

$$\{b \mid C \Longrightarrow b\} = \left\{\bigcup Cont_t \mid \forall t \in C,\ Cont_t \in \beta(t)\right\}.$$

The proof of Prop. 2.2.9 is shown in appendix A.3.

Intuitively, $\beta$ gives the set of all the choices available when spawning a continuation. To spawn a continuation for a thread the syntactic component of which is a choice $(P \oplus Q)$, we either spawn a continuation choice for $P$ or a continuation choice for $Q$. Spawning a continuation for a thread the syntactic component of which is a parallel composition $(P \mid Q)$, consists in choosing a continuation for $P$, choosing a continuation for $Q$, and spawning concurrently these two continuations.

Then, we can define the efficient semantics as the following transition system $(\mathscr{C}_0^{\mathrm{e}}(\mathscr{S}), \longrightarrow_{\mathrm{e}})$:

- $\mathscr{C}_0^{\mathrm{e}}(\mathscr{S}) = \beta(\mathscr{S}, \varepsilon, \emptyset)$
- $\forall \lambda \in \mathscr{L}^2, a \xrightarrow{\lambda}_{\mathrm{e}} b$ *if and only if* $\exists c,\ a \xrightarrow{\lambda}_{\mathrm{n}} c$ *and* $c \Longrightarrow b$.

An explicit definition of $(\mathscr{C}_0^{\mathrm{e}}(\mathscr{S}), \longrightarrow_{\mathrm{e}})$ is given in Fig. 2.8.

[4]We have also considered in [35, Sect. 3.3] restricting the set of the traces to those for which choices are made only when necessary, which comes down considering external choices instead of internal ones.

$$\begin{aligned}
\beta((\nu\, x)P, id, E) &= \beta\,(P, id, (E[x \mapsto (x, id)])) \\
\beta(\mathbf{0}, id, E) &= \{\emptyset\} \\
\beta(P \oplus Q, id, E) &= \beta(P, id, E) \cup \beta(Q, id, E) \\
\beta(P \mid Q, id, E) &= \{A \cup B \mid A \in \beta(P, id, E),\ B \in \beta(Q, id, E)\} \\
\beta(aP, id, E) &= \{\{(aP, id, E_{|fn(aP)})\}\}
\end{aligned}$$

(a) Extraction function

$$\mathscr{C}_0^{\mathrm{e}}(\mathscr{S}) = \beta(\mathscr{S}, \varepsilon, \emptyset)$$

(b) Initial confi gurations

$$\frac{\begin{cases} E_?(y) = E_!(x), \\ Ct_P \in \beta(P, id_?, E_?[y_i \mapsto E_!(x_i)]), \\ Ct_Q \in \beta(Q, id_!, E_!) \end{cases}}{C \cup \left\{ \begin{array}{l} (y?^i[\overline{y}]P, id_?, E_?), \\ (x!^j[\overline{x}]Q, id_!, E_!) \end{array} \right\} \xrightarrow{(i,j)}_{\mathrm{e}} (C \cup Ct_P \cup Ct_Q)}$$

$$\frac{\begin{cases} E_*(y) = E_!(x), \\ Ct_P \in \beta(P, N((i,j), id_*, id_!), E_*[y_i \mapsto E_!(x_i)]), \\ Ct_Q \in \beta(Q, id_!, E_!) \end{cases}}{C \cup \left\{ \begin{array}{l} (*y?^i[\overline{y}]P, id_*, E_*), \\ (x!^j[\overline{x}]Q, id_!, E_!) \end{array} \right\} \xrightarrow{(i,j)}_{\mathrm{e}} \left(C \cup \{(*y?^i[\overline{y}]P, id_*, E_*)\} \cup Ct_P \cup Ct_Q\right)}$$

(c) Communication rules

Figure 2.8: Efficient semantics.

#### 2.2.4.2 Correspondence

The following theorem establishes the correspondence between the standard and the efficient semantics:

**Theorem 2.2.10.** *For any initial non standard configuration $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$, there exists $k \in \mathbb{N}$ such that $\mathscr{S} \xrightarrow{\oplus^k}{}^* \Pi(C_0)$ and for all non standard configurations $C$ and for all word $u \in (\mathscr{L}^2)^*$ such that $C_0 \xrightarrow{u}{}_{\mathrm{e}}^* C$, we have:*

1. $\forall \lambda \in \mathscr{L}^2,\ C \xrightarrow{\lambda}{}_{\mathrm{e}} C' \implies \exists k \in \mathbb{N},\ \exists P,\ \Pi(C) \xrightarrow{\lambda} P \xrightarrow{\oplus^k}{}^* \Pi(C')$*;*
2. $\forall \lambda \in \mathscr{L}^2,\ \Pi(C) \xrightarrow{\lambda} P \implies \exists D, \begin{cases} C \xrightarrow{\lambda}{}_{\mathrm{e}} D \\ \textit{and} \begin{cases} \exists k > 0,\ P \xrightarrow{\oplus^k}{}^+ \Pi(D) \\ \textit{or } P \equiv \Pi(D). \end{cases} \end{cases}$

The proof of Thm. 2.2.10 is shown in appendix A.3.

**Remark 2.2.11.** *There is no bisimulation between the standard and the efficient semantics. We have restricted both the set of traces to those where choices are always performed before communications, and the set of states to those where no choice appears at the top level. Therefore, these restrictions do not change the properties we want to observe on mobile systems.*

# Chapter 3

# Dealing with location

Mobile ambients [17] are a model of mobile computation. It describes a set of *threads* which are distributed throughout hierarchically organized domains called *ambients*. Threads interact inside ambients which makes the ambients move, taking their content with them.

We first recall a standard semantics for mobile ambients in the Sect. 3.1. As in the case of the $\pi$-calculus (see also Chap. 2), we will notice that some interesting properties may not be specified, because there is link neither between the thread instances and the names of the ambients that they have declared, nor between the thread instances and the ambients that they have created. Thus we introduce in the Sect. 3.2 a refined semantics which is called the non standard semantics [37] in where this relationship is explicitly described.

## 3.1 Standard semantics for mobile ambients

We consider a lazy version of the mobile ambients in which replications are performed only when necessary. For the sake of simplicity, we restrict ourselves to name communications: only names and not capability paths may be communicated.

### 3.1.1 Syntax

Let $\mathscr{N}$ be a countable set of ambient names and $\mathscr{L}$ be a countable set of labels. We define the syntax of the mobile ambients in Fig. 3.1. Input action and restriction are the only name binders: in $(n)^l.P$, $!(n)^l.P$ and $(\nu\ n)P$, the occurrences of $n$ in $P$ are bound. Usual rules about scopes, substitution and $\alpha$-conversion apply. We denote by $fn(P)$ (resp. $bn(P)$) the set of the names that are free (resp. bound) in $P$. We locate each syntactic component of the system by placing distinct labels of $\mathscr{L}$.

$$
\begin{array}{rll}
n & \in \mathscr{N} & \text{(ambient name)} \\
l & \in \mathscr{L} & \text{(label)}
\end{array}
$$

$$
\begin{array}{rcll}
P, Q & ::= & (\nu\, n)P & \text{(restriction)} \\
 & | & \mathbf{0} & \text{(inactivity)} \\
 & | & P \mid Q & \text{(composition)} \\
 & | & n^{l}[P] & \text{(ambient)} \\
 & | & M & \text{(capability action)} \\
 & | & io & \text{(input/output action)}
\end{array}
$$

$$
\begin{array}{rcll}
M & ::= & in^{l}\ n.P & \text{(may enter } n\text{)} \\
 & | & out^{l}\ n.P & \text{(may exit } n\text{)} \\
 & | & open^{l}\ n.P & \text{(may open } n\text{)} \\
 & | & !open^{l}\ n.P & \text{(may duplicate itself before opening } n\text{)}
\end{array}
$$

$$
\begin{array}{rcll}
io & ::= & (n)^{l}.P & \text{(input action)} \\
 & | & !(n)^{l}.P & \text{(input action with replication)} \\
 & | & \langle n \rangle^{l} & \text{(asynchronous output action)}
\end{array}
$$

Figure 3.1: Syntax of mobile ambients.

### 3.1.2 Semantics

We now informally introduce the semantics of the *ambient* calculus. The thread $(\nu\, n)P$ creates a new ambient name $n$, the thread $P$ may use it to name new ambients and to operate on ambients named $n$. At first only the thread $P$ may interact and create ambient names $n$, but the thread $P$ may communicate the name $n$ to others threads, which then get the same rights over the name $n$. The thread $\mathbf{0}$ does nothing. In $(P \mid Q)$, $P$ and $Q$ are two concurrent threads which are located in the same ambient. They behave independently or interact. The thread $n^{l}[P]$ creates a new ambient named $n$ and launches the thread $P$ inside this new ambient. It is worth noting that several ambients may be named $n$ (also see [17][p:12]) and that two threads contained in distinct ambients may not interact even if their am-

$$\begin{array}{rcll}
P & \equiv & \mathrm{Q} \quad \text{if } P \sim_\alpha Q & (\alpha\text{-conversion}) \\
P \mid Q & \equiv & Q \mid P & \text{(Commutativity)} \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \text{(Associativity)} \\
P \mid \mathbf{0} & \equiv & P & \text{(Zero par)} \\
(\nu\, n)\mathbf{0} & \equiv & \mathbf{0} & \text{(Zero Res)} \\
(\nu\, n)(\nu\, m)P & \equiv & (\nu\, m)(\nu\, n)P & \text{(Swapping)} \\
(\nu\, n)(P \mid Q) & \equiv & P \mid ((\nu\, n)Q) \quad \text{if } n \notin fn(P) & \text{(Extrusion Par)} \\
(\nu\, n)(m^l[P]) & \equiv & m^l[(\nu\, n)P] \quad \text{if } n \neq m & \text{(Extrusion Amb)}
\end{array}$$

Figure 3.2: Congruence relation for mobile ambients.

$$\begin{array}{rcl}
n^i[in^k\; m.P \mid Q] \mid m^j[R] & \xrightarrow{in(i,j,k)} & m^j[n^i[P \mid Q] \mid R] \\
m^i[n^j[out^k\; m.P \mid Q] \mid R] & \xrightarrow{out(i,j,k)} & n^j[P \mid Q] \mid m^i[R] \\
open^i\; n.P \mid n^j[Q] & \xrightarrow{open(i,j)} & P \mid Q \\
!open^i\; n.P \mid n^j[Q] & \xrightarrow{open(i,j)} & P \mid Q \mid !open^i\; n.P \\
(n)^i.P \mid \langle m \rangle^j & \xrightarrow{com(i,j)} & P[n \leftarrow m] \\
!(n)^i.P \mid \langle m \rangle^j & \xrightarrow{com(i,j)} & P[n \leftarrow m] \mid !(n)^i.P
\end{array}$$

$$\frac{P \xrightarrow{\lambda} Q}{n^i[P] \xrightarrow{\lambda} n^i[Q]} \qquad \frac{P \xrightarrow{\lambda} Q}{(\nu\, n)P \xrightarrow{\lambda} (\nu\, n)Q} \qquad \frac{P \xrightarrow{\lambda} Q}{P \mid R \xrightarrow{\lambda} Q \mid R}$$

$$\frac{P' \equiv P,\; P \xrightarrow{\lambda} Q,\; Q \equiv Q'}{P' \xrightarrow{\lambda} Q'}$$

Figure 3.3: Reduction relation for mobile ambients.

bients have the same name. A thread may also perform a capability action. The thread $in^l\ n.P$ may take its surrounding ambient into an sibling ambient named $n$ before launching the continuation $P$. The thread $out^l\ n.P$ may take its surrounding ambient out of its parent ambient before launching the continuation $P$. The thread $open^l\ n.P$ may dissolve an ambient named $n$, before launching the continuation $P$. An ambient which is dissolved passes all its content to its parent ambient. The thread $!open^l\ n.P$ is like a resource: it may dissolve many ambients named $n$, it launches an instance of $P$ each time it dissolves an ambient. A thread may also perform a local communication: the thread $(n)^l.P$ waits for an ambient name. When it receives such a name, it binds the variable $n$ to the received name and launches the continuation $P$. The thread $!(n)^l.P$ is a resource, it may launch a continuation $P$ each time it receives an ambient name. The thread $\langle n\rangle^l$ sends the ambient name $n$, This sending is asynchronous, which means that there is no continuation.

The operational semantics is given by both a congruence in Fig. 3.2 and a reduction relation in Fig. 3.3. As for the $\pi$-calculus (Cf. Chap.2), the congruence relation allows threads to interact, while the reduction relation describes thread computations. Reduction rules are also graphically described in Figs. 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9. Some rules in the congruence relation make threads move inside the syntactic tree: they assert the associativity and commutativity of the parallel composition. Some others extend the scope of names to the threads they are communicated to or restrict the scope of the names that are used by an ambient that exits its parent ambient: $\alpha$-conversion solves conflicts between names, swapping selects the name the scope of which we wish to extend, and extrusion extends its scope to another thread or to a parent ambient. The reduction relation describes threads' interactions. An ambient $A$ may enter its sibling ambient $B$ named $n$ if it has a capability $in^l\ n.P$ to enter an ambient named $n$, the result is obtained by taking the migrating ambient $A$ inside its sibling ambient $B$ and by launching the continuation $P$ inside the ambient $A$. Conversely, an ambient $A$ may exit its parent ambient $B$ named $n$ if it has a capability $out^l\ n.P$ to exit an ambient named $n$, the result is given by taking the ambient $A$ inside the parent ambient of $B$ and by launching the continuation $P$ inside the ambient $A$. Until now, all the reductions that we have described allows the migration of ambients but not how ambients may communicate some control over the other ambients. Two distinct mechanisms allow these communications: dissolution and name passing. An ambient $A$ may dissolve an ambient $B$ named $n$ that is enclosed inside $A$ if it has a capability $open^l\ n.P$ to open an ambient named $n$, the result is obtained by passing all the content (ambients and threads) of $B$ inside $A$ and by launching the continuation $P$ inside $A$. When the dissolving thread is a resource, it is just syntactically replicated before performing the reduction; this way the resource is still available after the dissolution. Name passing is allowed when there are two concurrent

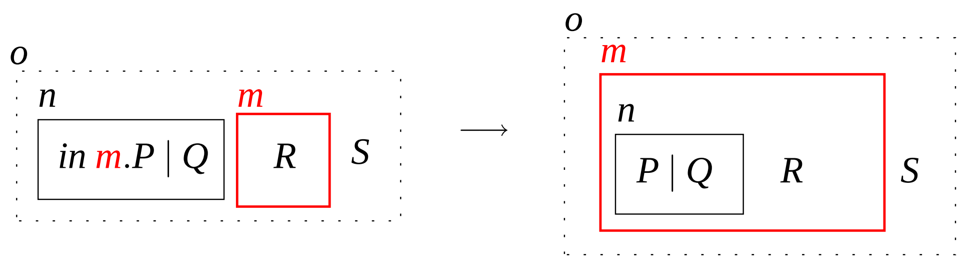


Figure 3.4: In migration.

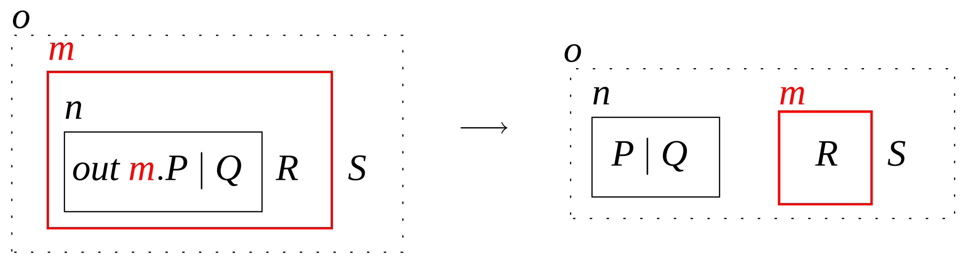


Figure 3.5: Out migration.

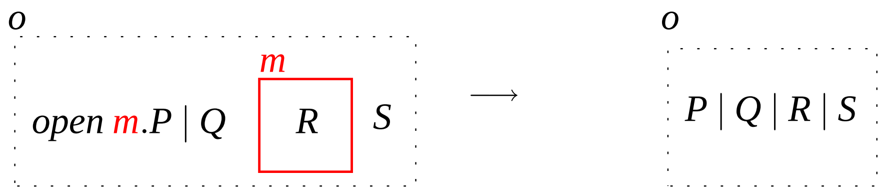


Figure 3.6: Dissolution.

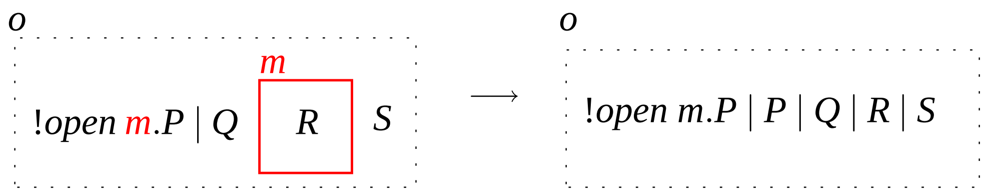


Figure 3.7: Dissolution with resource fetching.

threads in the same ambient. The results of such a communication are obtained by applying the substitution of the $\lambda$-calculus in the continuation of the message receiver. When the receiver is a resource, it is just syntactically replicated before performing the communication.

We have labeled each reduction step with a reduction name (*in*, *out*, *open* or *com*) and the labels of the threads and of the ambients that are involved in the interaction: this will allow us to relate the state of a system to the history of the computation steps that have led to this state.

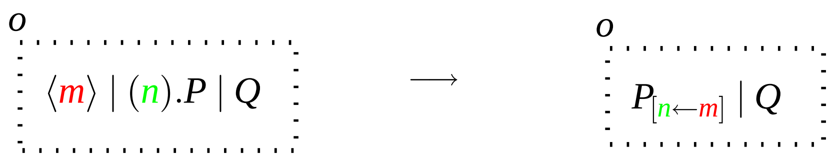


Figure 3.8: Communication.

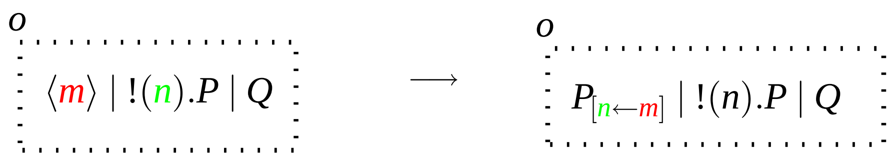


Figure 3.9: Communication with resource fetching.

**Example 3.1.1.** *We give in given in in Fig. 3.10 the definition of a system $\mathscr{S}$. This system $\mathscr{S}$ describes an* ftp *server. To make things clearer, public (or global) names are written in* roman, *all the other names are written in italic and we abstract away many computational aspects. A resource creates recursively an unbounded number of clients. Each client is described by a packet* $\mathbf{p}[]$ *which contains an ambient named* request. *This ambient contains the client's query* $\langle \mathbf{q} \rangle$. *When they are created, packets are located inside an ambient named* client. *At first, each packet exits the* client *ambient before entering the* server *ambient ; then this packet activates a pilot ambient* fwd *which communicates the packet name to the server. This communication creates a recursive instance of an ambient named* instance *which will process the packet. The* instance *ambient enters the packet, reads the request and sends it back inside an ambient named* answer. *At last, the packet exits the* server *ambient.*

*The definition of the system $\mathscr{S}$ in given in in Fig. 3.10. We give in Figs. 3.11*

$\nu\mathbf{Pub} := (\nu\ \mathrm{request})(\nu\ \mathrm{rec})(\nu\ \mathrm{client})(\nu\ \mathrm{server})(\nu\ \mathrm{fwd})(\nu\ \mathrm{instance})(\nu\ \mathrm{answer})$,
$C_1 := \mathrm{request}^{15}[\langle q\rangle^{16}]$, $\mathbf{C}_2 := open^{17}\mathrm{instance}$,
$\mathbf{C}_3 := out^{18}\mathrm{client}.in^{19}\mathrm{server}.\mathrm{fwd}^{20}[out^{21}p.\langle p\rangle^{22}]$,
$\mathbf{C} := (\nu\ q)(\nu\ p)p^{14}[\mathbf{C}_1 \mid \mathbf{C}_2 \mid \mathbf{C}_3] \mid \langle \mathrm{rec}\rangle^{23}$,
$\mathbf{I}_1 := \mathrm{answer}^{8}[\langle rep\rangle^{9}]$, $\mathbf{I}_2 := out^{10}\mathrm{server}.in^{11}\mathrm{client}$,
$\mathbf{I} := in^{5}k.open^{6}\mathrm{request}.(rep)^{7}(\mathbf{I}_1 \mid \mathbf{I}_2)$,
$\mathbf{S}_1 := !open^{2}\mathrm{fwd}$, $\mathbf{S}_2 := !(k)^{3}.\mathrm{instance}^{4}[\mathbf{I}]$, $\mathbf{S} := \mathrm{server}^{1}[\mathbf{S}_1 \mid \mathbf{S}_2]$,
$\mathscr{S} := (\nu\mathbf{Pub})\left(\mathbf{S} \mid \mathrm{client}^{12}\left[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{24}\right]\right)$.

Figure 3.10: An *ftp*-server written in the mobile ambients.

*and 3.12 a computation sequence that describes the behavior of the system $\mathscr{S}$.*

$$
\begin{array}{l}
(\nu\ \mathbf{Pub}) \\
\big(\mathbf{S} \mid \mathrm{client}^{12}\big[(\underline{x})^{13}.C \mid \langle \underline{\mathrm{rec}}\rangle^{24}\big]\big)
\end{array}
$$

$$\xrightarrow{com(13,24)}$$

$$
\begin{array}{l}
(\nu\ \mathbf{Pub}) \\
\left(\begin{array}{l}
\mathbf{S} \mid \\
\underline{\mathrm{client}}^{12}\left[\begin{array}{l}
!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23} \mid \\
(\nu\ q_1)(\nu\ p_1) \\
{p_1}^{14}\left[\begin{array}{l}
\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2 \mid \\
out^{18}\underline{\mathrm{client}}.in^{19}\mathrm{server}.\mathrm{fwd}^{20}\big[out^{21}p_1.\langle p_1\rangle^{22}\big]
\end{array}\right]
\end{array}\right]
\end{array}\right)
\end{array}
$$

$$\xrightarrow{out(12,14,18)}$$

$$
\begin{array}{l}
(\nu\ \mathbf{Pub}) \\
\left(\begin{array}{l}
\underline{\mathrm{server}}^{1}[\mathbf{S}_1 \mid \mathbf{S}_2] \mid \mathrm{client}^{12}\big[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23}\big] \mid \\
(\nu\ q_1)(\nu\ p_1){p_1}^{14}\left[\begin{array}{l}
\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2 \mid \\
in^{19}\underline{\mathrm{server}}.\mathrm{fwd}^{20}\big[out^{21}p_1.\langle p_1\rangle^{22}\big]
\end{array}\right]
\end{array}\right)
\end{array}
$$

$$\xrightarrow{in(14,1,19)}$$

$$
\begin{array}{l}
(\nu\mathbf{Pub})(\nu\ q_1)(\nu\ p_1) \\
\left(\begin{array}{l}
\mathrm{client}^{12}\big[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23}\big] \mid \\
\mathrm{server}^{1}\left[\begin{array}{l}
\mathbf{S}_1 \mid \mathbf{S}_2 \mid \\
{\underline{p}_1}^{14}\left[\begin{array}{l}
\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2 \mid \\
\mathrm{fwd}^{20}\big[out^{21}\underline{p}_1.\langle p_1\rangle^{22}\big]
\end{array}\right]
\end{array}\right]
\end{array}\right)
\end{array}
$$

$$\xrightarrow{out(14,20,21)}$$

$$
\begin{array}{l}
(\nu\mathbf{Pub})(\nu\ q_1)(\nu\ p_1) \\
\left(\begin{array}{l}
\mathrm{client}^{12}\big[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23}\big] \mid \\
\mathrm{server}^{1}\left[\begin{array}{l}
!open^{2}\underline{\mathrm{fwd}} \mid \mathbf{S}_2 \mid \underline{\mathrm{fwd}}^{20}\ \big[\langle p_1\rangle^{22}\big] \mid \\
{p_1}^{14}\big[\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2\big]
\end{array}\right]
\end{array}\right)
\end{array}
$$

$$\xrightarrow{open(2,20)}$$

$$
\begin{array}{l}
(\nu\mathbf{Pub})(\nu\ q_1)(\nu\ p_1) \\
\left(\begin{array}{l}
\mathrm{client}^{12}\big[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23}\big] \mid \\
\mathrm{server}^{1}\left[\begin{array}{l}
\mathbf{S}_1 \mid !(\underline{k})^{3}.\mathrm{instance}^{4}[\mathbf{I}] \mid \langle \underline{p}_1\rangle^{22} \mid \\
{p_1}^{14}\big[\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2\big]
\end{array}\right]
\end{array}\right)
\end{array}
$$

$$\xrightarrow{com(3,22)}$$

$$
\begin{array}{l}
(\nu\mathbf{Pub})(\nu\ q_1)(\nu\ p_1) \\
\left(\begin{array}{l}
\mathrm{client}^{12}\big[!(x)^{13}.C \mid \langle \mathrm{rec}\rangle^{23}\big] \mid \\
\mathrm{server}^{1}\left[\begin{array}{l}
\mathbf{S}_1 \mid \mathbf{S}_2 \mid {p_1}^{14}\big[\mathrm{request}^{15}\big[\langle q_1\rangle^{16}\big] \mid \mathbf{C}_2\big] \mid \\
\mathrm{instance}^{4}\big[in^{5}p_1.open^{6}\mathrm{request}.(rep)^{7}(\mathrm{I}_1|\mathrm{I}_2)\big]
\end{array}\right]
\end{array}\right)
\end{array}
$$

Figure 3.11: Computation sequence.

$$
\begin{array}{l}
(\nu\, \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12}\left[\,!(x)^{13}.C \mid \langle \text{rec}\rangle^{23}\right] \mid \\ \text{server}^{1} \left[ \begin{array}{l} \mathbf{S}_1 \mid \mathbf{S}_2 \mid \underline{p_1}^{14}\left[\text{request}^{15}\left[\langle q_1\rangle^{16}\right] \mid \mathbf{C}_2\right] \mid \\ \text{instance}^{4}\left[\mathit{in}^{5}\underline{p_1}.\mathit{open}^{6}\text{request}.(\mathit{rep})^{7}(\mathrm{I}_1|\mathrm{I}_2)\right] \end{array} \right] \end{array} \right) \\
\xrightarrow{\mathit{in}(4,14,5)} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12}\left[\,!(x)^{13}.C \mid \langle \text{rec}\rangle^{23}\right] \mid \\ \text{server}^{1} \left[ \begin{array}{l} \mathbf{S}_1 \mid \mathbf{S}_2 \mid \\ p_1{}^{14} \left[ \begin{array}{l} \text{request}^{15}\left[\langle q_1\rangle^{16}\right] \mid \mathit{open}^{17}\underline{\text{instance}} \mid \\ \underline{\text{instance}}^{4}[\mathit{open}^{6}\text{request}.(\mathit{rep})^{7}(\mathrm{I}_1|\mathrm{I}_2)] \end{array} \right] \end{array} \right] \end{array} \right) \\
\xrightarrow{\mathit{open}(17,4)} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12}\left[\,!(x)^{13}.C \mid \langle \text{rec}\rangle^{23}\right] \mid \\ \text{server}^{1} \left[ \begin{array}{l} \mathbf{S}_1 \mid \mathbf{S}_2 \mid \\ p_1{}^{14} \left[ \begin{array}{l} \underline{\text{request}}^{15}\left[\langle \underline{q_1}\rangle^{16}\right] \mid \\ \mathit{open}^{6}\underline{\text{request}}.(\underline{\mathit{rep}})^{7}(\mathrm{I}_1 \mid \mathrm{I}_2) \end{array} \right] \end{array} \right] \end{array} \right) \\
\longrightarrow^{*} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12}\left[\,!(x)^{13}.C \mid \langle \text{rec}\rangle^{23}\right] \mid \\ \underline{\text{server}}^{1} \left[ \begin{array}{l} \mathbf{S}_1 \mid \mathbf{S}_2 \mid \\ p_1{}^{14}\left[\text{answer}^{8}\left[\langle q_1\rangle^{9}\right] \mid \mathit{out}^{10}\underline{\text{server}}.\mathit{in}^{11}\text{client}\right] \end{array} \right] \end{array} \right) \\
\xrightarrow{\mathit{out}(1,14,10)} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \underline{\text{client}}^{12}\left[\,!(x)^{13}.C \mid \langle \text{rec}\rangle^{23}\right] \mid \\ \text{server}^{1}\,[\mathbf{S}_1 \mid \mathbf{S}_2] \mid \\ p_1{}^{14}\left[\text{answer}^{8}\left[\langle q_1\rangle^{9}\right] \mid \mathit{in}^{11}\underline{\text{client}}\right] \end{array} \right)] \\
\xrightarrow{\mathit{in}(14,12,11)} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12} \left[ \begin{array}{l} !(x)^{13}.C \mid \langle \text{rec}\rangle^{23} \mid \\ p_1{}^{14}\left[\text{answer}^{8}\left[\langle q_1\rangle^{9}\right]\right] \end{array} \right] \mid \\ \text{server}^{1}\,[\mathbf{S}_1 \mid \mathbf{S}_2] \end{array} \right) \\
\longrightarrow^{*} \\
(\nu \mathbf{Pub})(\nu\, q_1)(\nu\, p_1) \\
\quad \left( \begin{array}{l} \text{client}^{12} \left[ \begin{array}{l} !(x)^{13}.C \mid \langle \text{rec}\rangle^{23} \mid \\ p_1{}^{14}\left[\text{answer}^{8}\left[\langle q_1\rangle^{9}\right]\right] \\ p_2{}^{14}\left[\text{answer}^{8}\left[\langle q_2\rangle^{9}\right]\right] \end{array} \right] \mid \\ \text{server}^{1}\,[\mathbf{S}_1 \mid \mathbf{S}_2] \end{array} \right)
\end{array}
$$

Figure 3.12: Computation sequence(continued).

## 3.2 Non standard semantics

The non standard semantics is a refined one with explicit substitution. It restores the link between the recursive instances of threads and the objects they have created (i.e. the names they have declared and the ambients they have activated). Following $D\pi$ [45] style, we describe a mobile system with a set of threads tagged with a location marker. Furthermore, the embedding structure of the ambients imposes a description of the hierarchical tree of the administrative domains (or ambients). This is given by a set of activated ambients[1] (seen as locations) tagged with location markers specifying their surrounding ambient. We assume that a system is run inside a top level ambient which has no location. The link between threads and the ambient names they have declared is made explicit by tagging each thread by an unambiguous history marker allocated at its creation. Then, each new ambient name is tagged with the history marker of the thread which has declared it. Thus, we restore the link between threads and the ambients they have activated by tagging each activated ambient with the history marker of the thread which has activated it.

Let $\mathscr{S}$ be a closed mobile system in the *ambient* calculus. We assume without any loss of generality that two name binders ($\nu\ n$ or $(n)$) are never used to bind the same ambient name. *History markers* are binary trees the node of which are labeled with elements of $\mathscr{L}^2$ and the leaves of which are not labeled. The tree having a node labeled $\lambda$, a left sibling $t_1$ and a right one $t_2$ is denoted by $N(\lambda, t_1, t_2)$. Moreover, leaves are denotes by $\varepsilon$. We denote by *Id* the set of the history markers. *Ambient names* are described by a pair $(n, id)$ where $n$ specifies which restriction $(\nu\ n)$ has created it while *id* is the history marker of the thread which has declared this name. *Activated ambients* are identified by a pair $(i, id)$ where $i$ is the label of the ambient constructor which has activated the ambient while *id* is the history marker of its activator[2]. The top level ambient is denoted by the pair $(\text{top}, \varepsilon)$ (we assume that $\text{top} \in \mathscr{L}$ has not been used for labeling $\mathscr{S}$ yet). *Location markers* are pairs $(i, id)$, too. A location marker refers to the ambient where a process is spawned.

A *non standard configuration* [74, 36] is a set of thread instances, where a thread instance is a tuple composed of a syntactic component, a history marker, a location marker and an environment. The syntactic component is either a syntactic copy of a thread of $\mathscr{S}$ or an activated ambient denoted by $n^i[\bullet]$. The history marker is unambiguously allocated at the thread creation. The location marker indicates where the thread is run. The environment specifies the origin of the free syntactic

[1]Also called privileged ambients in [17].

[2]An ambient may not be identifi ed by its ambient name because two distinct activated ambients may have the same name [17, p:12].

ambient names of the syntactic component.

**Example 3.2.1.** *We give here the non standard configuration reached after completing two sessions of our server*[3]*:*

$$\begin{cases}
(\mathrm{server}^1[\bullet], \varepsilon, (\mathrm{top}, \varepsilon), \emptyset) \\
(\mathrm{answer}^8[\bullet], id'_0, (14, id_0), \emptyset) \\
(\mathrm{answer}^8[\bullet], id'_1, (14, id_1), \emptyset) \\
(p^{14}[\bullet], id_0, (\mathrm{top}, \varepsilon), [p \mapsto (p, id_0)]) \\
(p^{14}[\bullet], id_1, (\mathrm{top}, \varepsilon), [p \mapsto (p, id_1)]) \\
(!(x)^{13}.\boldsymbol{C}, \varepsilon, (\mathrm{top}, \varepsilon), \emptyset) \\
(\langle \mathrm{rec} \rangle^{23}, id_1, (\mathrm{top}, \varepsilon), \emptyset) \\
(\boldsymbol{S}_1, \varepsilon, (1, \varepsilon), \emptyset) \\
(\boldsymbol{S}_2, \varepsilon, (1, \varepsilon), \emptyset) \\
(\langle rep \rangle^{10}, id'_0, (8, id'_0), [rep \mapsto (q, id_0)]) \\
(\langle rep \rangle^{10}, id'_1, (8, id'_1), [rep \mapsto (q, id_1)])
\end{cases}
\quad \textit{where:} \quad
\begin{cases}
id_0 = N((13,24), \varepsilon, \varepsilon) \\
id_1 = N((13,23), \varepsilon, id_0) \\
id'_0 = N((3,22), \varepsilon, id_0) \\
id'_1 = N((3,22), \varepsilon, id_1)
\end{cases}$$

*The top five instances describe the hierarchic structure of nested ambients, the others describe the thread distribution. Location markers allow in reconstructing the following ambient:*

$$(\nu\, \overline{n}) \quad {}^{(\mathrm{top},\varepsilon)}\left[\begin{array}{l}
!(x)^{(13,\varepsilon)}.\boldsymbol{C} \mid \langle \mathrm{rec} \rangle^{(23,id_1)} \mid \mathrm{server}^{(1,\varepsilon)}[\boldsymbol{S}_1 \mid \boldsymbol{S}_2] \mid \\
(p, id_0)^{(14,id_0)}[\mathrm{answer}^{(8,id'_0)}[\langle (q, id_0) \rangle^{(10,id'_0)}]] \mid \\
(p, id_1)^{(14,id_1)}[\mathrm{answer}^{(8,id'_1)}[\langle (q, id_1) \rangle^{(10,id'_1)}]]
\end{array}\right]$$

*in which ambients, ambient names and threads are stamped with their own markers. Thanks to name markers, we avoid conflict between ambient names. So we may extrude their declaration inside the top level ambient. In this way, the shortcut $(\nu\overline{n})$ denotes the declaration of all the ambient names of the configuration. It appears explicitly that, in each packet, both the name of the packet and that contained in the "answer" ambient embedded in the packet have been declared by the same recursive instance of the resource $!(x).\mathbf{C}$. This means that the answer of a query is sent to the good client.*

*We shall remark that, since $\mathscr{S}$ has no nested resources, markers are all sequences instead of trees.* □

The non standard semantics is given by both an initial non standard configuration in Fig. 3.14 and a reduction relation (migration rules are given in Figs. 3.15 and 3.16, dissolution rules in Figs. 3.17 and 3.18 and communication rules in

[3]We do not fi gure the origin of public names.

$$\begin{array}{rcl}
\beta(n^i[P],id,loc,E) & = & \beta(P,id,(i,id),E)\cup\{(n^i[\bullet],id,loc,[n\mapsto E(n)])\}\\
\beta(P\mid Q,id,loc,E) & = & \beta(P,id,loc,E)\cup\beta(Q,id,loc,E)\\
\beta((\nu\, n)P,id,loc,E) & = & \beta\,(P,id,loc,(E[n\mapsto (n,id)]))\\
\beta(M,id,loc,E) & = & \{(M,id,loc,E_{|fn(M)})\}\\
\beta(io,id,loc,E) & = & \{(io,id,loc,E_{|fn(io)})\}\\
\beta(\mathbf{0},id,loc,E) & = & \emptyset
\end{array}$$

Figure 3.13: Extraction function.

Figs. 3.19 and 3.20). Their definitions use the extraction function $\beta$ defined in Fig. 3.13. Given a continuation $P$, a history marker *id*, a location marker *loc* and an environment $E$, $\beta(P,id,loc,E)$ gives the set of all the thread instances that must be spawned to simulate the computation of the process $E(P)$ identified with the marker *id*, in the ambient denoted by *loc*. It especially deals with new ambient name declaration and new ambient activation.

We informally describe the non standard semantics. For the sake of the brevity, we only detail the non standard *in* migration rule. *in* migration rule involves two distinct ambients $\lambda$, $\mu$ and a thread $\psi$. They are respectively denoted by three threads $(n^i[\bullet],id_1,loc_1,E_1)$, $(m^j[\bullet],id_2,loc_2,E_2)$ and $(in^k o.P,id_3,loc_3,E_3)$. The *in* migration rule is enabled if and only if the two ambients are located in the same ambient (this yields the constrain $loc_1 = loc_2$), the thread is located in the first ambient (this yields $loc_3 = (i,id_1)$) and the thread capability may interact with the name of the second ambient (this is encoded by the constrain $E_2(m) = E_3(o)$). The result of such a migration is that the first ambient moves inside the second one (its location is just replaced by $(j,id_2)$). All it content is taken with it (this does change neither their location markers, nor their environments), but the thread $\psi$ is executed and its continuation is spawned inside the first ambient ($\psi$ is replaced by $\beta(P,id_3,loc_3,E_{3|fn(P)})$). The *out* migration is simulated in the same way. The ambient dissolution is a bit much complex since all the locations of the dissolved ambient content are changed. We shall notice that each time a resource is fetched, a new history marker is deterministically allocated: it is given by $N((i,j),id_i,id_j)$ where $i$ is the label of the resource, $id_i$ is the history marker of the resource, $j$ is the label of the thread which enforces the resource fetching and $id_j$ is the history marker of this thread. We do not need a congruence relation because our set-based representation of configurations makes structural congruence rules useless and the use of history markers avoids conflicts between ambient names.

Standard and non standard semantics are strongly bisimilar. The proof relies on that non standard computations may not yield conflicts between history markers. Moreover, according to the following proposition, we can simplify[4] the shape

[4]We have reversed markers in order to restore chronological order (so that the past is on the

$$C_0(\mathscr{S}) = \beta(\mathscr{S}, \varepsilon, (\mathrm{top}, \varepsilon), \emptyset).$$

Figure 3.14: Initial configuration.

If $C$ is a non standard configuration,
if $\lambda$, $\mu$ and $\psi$ (with $\lambda \neq \mu$) are three threads

$$\text{with } \begin{cases} \lambda = \left(n^i[\bullet], id_1, loc_1, E_1\right), \\ \mu = \left(m^j[\bullet], id_2, loc_2, E_2\right), \\ \psi = \left(in^k o.P, id_3, loc_3, E_3\right), \end{cases}$$

$$\text{such that } \begin{cases} loc_1 = loc_2, \\ loc_3 = (i, id_1), \\ E_2(m) = E_3(o), \end{cases}$$

then:

$$C \cup \{\lambda, \mu, \psi\} \xrightarrow{in(i,j,k)}_{\mathrm{ns}} C \cup \left\{\mu, \left(n^i[\bullet], id_1, (j, id_2), E_1\right)\right\} \cup \beta\left(P, id_3, loc_3, E_{3|fn(P)}\right).$$

Figure 3.15: Non standard in migration.

If $C$ is a non standard configuration,
if $\lambda$, $\mu$ and $\psi$ are three threads

$$\text{with } \begin{cases} \lambda = \left(m^i[\bullet], id_1, loc_1, E_1\right), \\ \mu = \left(n^j[\bullet], id_2, loc_2, E_2\right), \\ \psi = \left(out^k o.P, id_3, loc_3, E_3\right), \end{cases}$$

$$\text{such that } \begin{cases} loc_2 = (i, id_1), \\ loc_3 = (j, id_2), \\ E_1(m) = E_3(o), \end{cases}$$

then:

$$C \cup \{\lambda, \mu, \psi\} \xrightarrow{out(i,j,k)}_{\mathrm{ns}} C \cup \left\{\lambda; \left(n^j[\bullet], id_2, loc_1, E_2\right)\right\} \cup \beta\left(P, id_3, loc_3, E_{3|fn(P)}\right).$$

Figure 3.16: Non standard out migration.

If $C$ is a non standard configuration,
if $\lambda$ and $\mu$ are two threads,
with $\begin{cases} \lambda = \left(open^i m.P, id_1, loc_1, E_1\right), \\ \mu = \left(n^j[\bullet], id_2, loc_2, E_2\right), \end{cases}$
such that $\begin{cases} loc_1 = loc_2, \\ E_1(m) = E_2(n), \end{cases}$
then:

$$C \cup \{\lambda, \mu\} \xrightarrow{open(i,j)}_{\text{ns}} (C \setminus A) \cup \beta\left(P, id_1, loc_1, E_{1|fn(P)}\right) \cup A',$$

where $\begin{cases} A = \{(a, id, loc, E) \in C \mid loc = (j, id_2)\} \\ A' = \{(a, id, loc_2, E) \mid (a, id, (j, id_2), E) \in C\}. \end{cases}$

Figure 3.17: Non standard dissolution.

If $C$ is a non standard configuration,
if $\lambda$ and $\mu$ are two threads,
with $\begin{cases} \lambda = \left(!open^i m.P, id_1, loc_1, E_1\right), \\ \mu = \left(n^j[\bullet], id_2, loc_2, E_2\right), \end{cases}$
such that $\begin{cases} loc_1 = loc_2 \\ E_1(m) = E_2(n), \end{cases}$
then:

$$C \cup \{\lambda; \mu\} \xrightarrow{ropen(i,j)}_{\text{ns}} ((C \cup \{\lambda\}) \setminus A) \cup \beta\left(P, N((i,j), id_1, id_2), loc_1, E_{1|fn(P)}\right) \cup A',$$

where $\begin{cases} A = \{(a, id, loc, E) \in C \mid loc = (j, id_2)\} \\ A' = \{(a, id, loc_2, E) \mid (a, id, (j, id_2)), E) \in C\}. \end{cases}$

Figure 3.18: Non standard replicated dissolution.

If $C$ is a non standard configuration,
if $\lambda$ and $\mu$ are two threads,
with $\begin{cases} \lambda = ((n)^i.P, id_1, loc_1, E_1), \\ \mu = (\langle m\rangle^j, id_2, loc_2.E_2), \end{cases}$
such that $loc_1 = loc_2$,
then:

$$C \cup \{\lambda, \mu\} \xrightarrow{com(i,j)}_{\text{ns}} C \cup \beta\left(P, id_1, loc_1, E_1\left[n \mapsto E_2(m)\right]_{|fn(P)}\right).$$

Figure 3.19: Non standard communication.

If $C$ is a non standard configuration,
if $\lambda$ and $\mu$ are two threads,
with $\begin{cases} \lambda = (!(n)^i.P, id_1, loc_1, E_1), \\ \mu = (\langle m\rangle^j, id_2, loc_2.E_2), \end{cases}$
such that $loc_1 = loc_2$,
then:

$$C \cup \{\lambda, \mu\} \xrightarrow{fetch(i,j)}_{\text{ns}} C \cup \{\lambda\} \cup \beta\left(P, N((i,j), id_1, id_2), loc_1, E_1\,[n \mapsto E_2(m)]_{|fn(P)}\right).$$

Figure 3.20: Non standard resource fetching.

of the history markers without losing the consistency of our semantics.

$$\phi_1 : \begin{cases} Id & \rightarrow & (\mathscr{L}^2)^\star \\ N(a,b,c) & \mapsto & \phi_1(c).a \\ \varepsilon & \mapsto & \varepsilon \end{cases} \qquad \phi_2 : \begin{cases} Id & \rightarrow & \mathscr{L}^\star \\ N((i,j),b,c) & \mapsto & \phi_2(c).j \\ \varepsilon & \mapsto & \varepsilon \end{cases}$$

**Proposition 3.2.2.** *Let $\phi$ be $\phi_1$ or $\phi_2$ and $C_0 \longrightarrow_{\text{ns}} \ldots \longrightarrow_{\text{ns}} C_n$ be a non standard computation sequence, where $C_0 = C_0(\mathscr{S})$. For all $i, j \in [|0,n|]$, $(p, id, loc, E) \in C_i$ and $(p', id', loc', E') \in C_j$, such that $\phi(id) = \phi(id')$ then $id = id'$.*

Such simplifications allow us to reduce the cost of our analysis, but also lead to a loss of accuracy, since they merge information related to distinct computation sequences of the system.

## 3.3 Related works

Several semantics that describe an ambient configuration by a set of threads have been proposed simultaneously to ours. A first application is the design of abstract machine for implementing ambient-like calculus: Sangiorgi and Valente have proposed such a semantics in [71]. Their purpose is to describe a distributed abstract machine for implementing the *safe ambients* [71] which are a variation of the mobile ambient. In [66], Phillips, Yoshida, and Eisenbach have later exploited similar ideas to derive a distributed abstract machine for the *boxed ambients* [13, 12]. A second application is in static analysis. Maffeis and Levi have proposed in [50, 51] a normal semantics. However there is a difference between our semantics and these others: Only our semantics relates precisely threads and both the names that they declare and the ambients that they create.

left and the future in on the right)

# Chapter 4

# Meta language

In this chapter, we propose a generic framework for describing the non standard semantics of models for mobile systems. Each encoded model is defined by some generic partial interactions that are computed by threads and by some generic rules which describe the behavior of threads when they synchronize the computation of their partial interactions. Then, the syntax of a system is given by a set of program points and a static description of the partial interactions that may be performed at each program point. The state of a system is a set of thread instances which are obtained by associating a program point with a marker and an environment. The operational semantics can then be generically derived from the language. We stress the point that our purpose is not to propose an abstract machine for implementing concrete interpreters. Our meta language helps in expressing semantics which can be easily analyzed.

We first give the meta syntax of languages in Sect. 4.1. Then we define the syntax of the mobile systems which are expressed in these languages in Sect. 4.2. System states are described in Sect. 4.3. The generic operational semantics is described in Sect. 4.4. In Sect. 4.5, we give sufficient conditions over the system syntax to ensure the freshness of allocated markers. We encode some models in our meta language in Chap. 5.

## 4.1 Meta syntax

In this section, we define the meta syntax of a language. This meta syntax is defined by a set of partial interactions that may be computed by threads, and by a set of rules that describe the generic interactions between some threads that synchronize the computation of their partial interactions.

### 4.1.1 Partial interactions

We first introduce a set $\mathscr{A}$ of generic partial interaction names. A map $arity \in \mathscr{A} \rightarrow \mathbb{N}^2$ maps each partial interaction name into two numbers of parameters. The first one is the number of the parameters that must be taken into account when enabling a global interaction. The second one denotes the number of the variables that will be bound to some value communicated by some other threads during the global interaction. For instance, in the case of the $\pi$-calculus, the reception of a channel name pair will be associated with the couple $(1,2)$, the first number 1 refers to the name of the channel on which the reception is performed, while the second number 2 indicates that two variables will be bound to some passed channel names. Another map $type \in \mathscr{A} \rightarrow \{computation; migration; replication\}$ associates a type with each partial interaction name. This type constrains the behavior of the threads when they compute such a partial interaction. Computing a partial interaction may consist in launching a continuation (i.e *computation*); it may also consist in changing the environment of a thread (i.e. *migration*); a thread may also replicate itself before launching a continuation (i.e. *replication*). These types have two purposes. First, they describe whether a computing thread is still available after the computation step. More precisely, a partial interaction of type *replication* does not consume its thread, whereas the others do. Second, they restrict the kind of partial interaction that may be involved in a computation step in order to prove the freshness of allocated markers: We will make the assumption that when in a global interaction, one partial interaction is a *replication*, then at least one partial interaction is a *computation*. This way, a thread may only be replicated when another one is consumed. This allows an unambiguous encoding of thread histories.

### 4.1.2 Rules

A global interaction involves several threads which enable the correct partial interactions and which share compatible interfaces. In such a case, the interaction may be performed. As a result, threads may pass some values to each other and launch new threads.

#### 4.1.2.1 Threads and capabilities

Global interactions are symbolically described by generic reduction rules. Each rule involves a given number $n$ of threads. These threads are indexed over the integers that range between 1 and $n$. Each index $i$ is associated with a partial interaction name via a map *components*. To perform a global interaction, a mobile system must satisfy a first condition: it must contain $n$ threads and each of these threads must exhibit the capability to process the corresponding partial interaction. Then, a mobile system must check some constraints about the values that are bound to the variables of the interacting threads.

#### 4.1.2.2 Formal variables

For that purpose, we introduce a set of formal variables. Each thread involved in the interaction is associated with two sequences of parameter variables. The first sequence is the sequence of the variables the value of which is involved in the interaction (i.e. the variables, including thread identities, that are associated with passed values or the variables that are associated with some values that allow for the thread synchronization). The second sequence is the sequence of the variables that are bound during the computation step to some values passed by the other threads.

We introduce the three sets of formal variables that denote these parameter variables and the threads' identities as follows:

1. The set $\mathcal{V}_f^I$ of the variables of the form $I^k$, where $I^k$ denotes the identity of the $k$-th interacting thread;

2. The set $\mathcal{V}_f^X$ of the variables of the form $X_l^k$, where $X_l^k$ denotes the value that is associated with the $l$-th parameter variable of the $k$-th thread of the interaction;

3. The set $\mathcal{V}_f^Y$ of the variables of the form $Y_l^k$, where $Y_l^k$ denotes the variable that is associated with the $l$-th variable of the $k$-th thread of the interaction which is bound during the interaction.

**Example 4.1.1.** *In the case of the $\pi$-calculus, Let $t_1 = (p_1, id_1, E_1)$ and $t_2 = (p_2, id_2, E_2)$ be two threads such that $t_1$ is a thread at program point $a?[x].P$ and $t_2$ is a thread at program point $b![y].Q$. The thread $t_1$ is associated with the sequence $[a]$ of variable parameters and with the sequence $[x]$ of bound variables. The thread $t_2$ is associated with the sequence $[b;y]$ of parameters and with the empty sequence of bound variables. Thus, the communication between $t_1$ and $t_2$ is associated with the following variables:*

- *The variable $I^1$ denotes the value $(p_1, id_1)$ and the variable $I^2$ denotes the value $(p_2, id_2)$, which are respectively the identity of the threads $t_1$ and $t_2$;*

- *The variable $X_1^1$ denotes the value $E_1(a)$;*

- *The variable $X_1^2$ denotes the value $E_2(b)$ and the variable $X_2^2$ denotes the value $E_2(y)$;*

- *The variable $Y_1^1$ denotes the variable x that will be bound to the value $E_2(y)$ during the communication computation.*

#### 4.1.2.3 Synchronization

Before interacting, threads must check some properties about their interface. These properties are given by an equivalence class between the formal variables in the set $(\mathcal{V}_f^X \cup \mathcal{V}_f^I)$. This equivalence class is described by a subset of $(\mathcal{V}_f^X \cup \mathcal{V}_f^I)^2$, where each constraint $(A,B)$ means that the interaction may be performed only if the formal variables $A$ and $B$ are associated with the same value at run-time.

**Example 4.1.2.** *In the case of the $\pi$-calculus, let $t_1$ and $t_2$ be the same threads as in Ex. 4.1.1. The compatibility between the threads $t_1$ and $t_2$ can be described by the equivalence class that is generated by the constraint $(X_1^1, X_1^2)$. This constraint means that the communication is enabled only if $E_1(a) = E_2(a)$.*

#### 4.1.2.4 Local communication

When an interaction is performed, some threads pass some of their values to the other threads. These communications are described by a partial map from $\mathcal{V}_f^Y$ into the set $(\mathcal{V}_f^X \cup \mathcal{V}_f^I)$. The fact that a variable $Y_l^k$ is associated with the variable $A$ means that, the $l$-th bound variable of the $k$-th thread will be associated at run-time with the value that is associated with the formal variable $A$.

**Example 4.1.3.** *In the case of the $\pi$-calculus, let $t_1$ and $t_2$ be the same threads as in Example 4.1.1. The result of the communication is that a variable x is declared in the thread P and associated with the value $E_2(b)$. This value passing is encoded by the partial map $[Y_1^1 \mapsto X_2^2]$.*

#### 4.1.2.5 Global substitutions

A computation step may destroy a thread and redirect any reference to this thread in the whole system to another value. For instance, in the case of mobile ambients, any component of a dissolved ambient is relocated to the parent of the dissolved ambient. This mechanism goes against the intuition of system distribution and must be handled very carefully, since it allows broadcast-*like* communications, where a message is simultaneously received by any thread that is waiting for a message on a given channel. Nevertheless, in the case of the ambient calculus, the mechanism is distributive-safe, because it only allows updating the location of some threads that are located in the same ambient. These global substitutions are encoded by a partial map from the set $\mathcal{V}_f^I$ into the set $(\mathcal{V}_f^X \cup \mathcal{V}_f^I)$.

**Example 4.1.4.** *In the case of the ambient calculus, let $t_1$ be a thread that denotes an ambient, and let $t_2$ be a thread that has the capability to dissolve this ambient. The variable $I^1$ denotes the identity of the ambient, while the variable $X_1^1$ denotes the location of the ambient. When the ambient is dissolved, any thread located inside the ambient is taken to the parent ambient, which is described by the following mapping $[I^1 \mapsto X_1^1]$.*

#### 4.1.2.6 Formal rule

This yields the following definition:

**Definition 4.1.5.** A rule is given by a tuple:

$$\mathscr{R} = (n, \mathit{components}, \mathit{compatibility}, \mathit{v\text{-}passing}, \mathit{broadcast}),$$

where:

- the integer $n$ is the arity of the rule: it is the number of threads involved in the interaction;
- the function *components* $\in [\![1;n]\!] \rightarrow \mathscr{A}$ maps each integer to the name of a partial interaction, each thread involved in the interaction must have the capability to process the corresponding partial interaction;

  For any $k$ such that $1 \leq k \leq n$, we denote $(p_k, b_k) = \mathit{arity}(\mathit{components}(k))$. We also introduce three sets of formal variables:

  1. the set $\mathscr{V}_{\mathscr{R}}^I \subseteq \mathscr{V}_f^I$ of the variables $I^k$ where $1 \leq k \leq n$;
  2. the set $\mathscr{V}_{\mathscr{R}}^X \subseteq \mathscr{V}_f^X$ of the variables $X_l^k$ where $1 \leq k \leq n$ and $1 \leq l \leq p_k$;
  3. the set $\mathscr{V}_{\mathscr{R}}^Y \subseteq \mathscr{V}_f^Y$ of the variables $Y_l^k$ where $1 \leq k \leq n$ and $1 \leq l \leq b_k$;

- the relation *compatibility* $\in \left(\mathscr{V}_{\mathscr{R}}^I \cup \mathscr{V}_{\mathscr{R}}^X\right)^2$ describes synchronization conditions among the interfaces of the interacting threads;
- the map *v-passing* $\in \mathscr{V}_{\mathscr{R}}^Y \rightarrow \left(\mathscr{V}_{\mathscr{R}}^I \cup \mathscr{V}_{\mathscr{R}}^X\right)$ describes local values passing;
- the partial map *broadcast* from $\mathscr{V}_{\mathscr{R}}^I$ into $\left(\mathscr{V}_{\mathscr{R}}^I \cup \mathscr{V}_{\mathscr{R}}^X\right)$ describes substitutions that apply within the whole system.

### 4.1.3 Well-formedness conditions

In order to ensure the freshness of allocated markers, we require some extra assumptions about formal rules. A new marker is computed each time a resource is fetched. We require that any rule may involve at most one replication. Moreover we require that any rule that involves a replication also involves a regular computation. This way, a new marker can only be allocated by consuming a thread instance the identity of which is explicitly described in this new marker. So a marker can only be computed once in a computation sequence.

Without any loss of expressibility, we require that, at each replication, the replicated thread is always the first thread and that the second thread performs a regular computation:

**Definition 4.1.6 (well-formed rule).**
Let $\mathscr{R} = (n, \mathit{components}, \mathit{compatibility}, \mathit{v\text{-}passing}, \mathit{broadcast})$ be a formal rule. The rule $\mathscr{R}$ is well-formed if and only if: if there exists an integer $i \in [\![1;n]\!]$ such that $\mathit{type}(\mathit{components}(i)) = \mathit{replication}$, then $i = 1$ and $\mathit{type}(\mathit{components}(2)) = \mathit{computation}$.

## 4.2 System syntax

We now describe the syntax of mobile systems.

### 4.2.1 Labeling

Let $\mathscr{L}$ be a finite set of labels. These labels are used in tagging program points and values. Thus, let $\mathscr{L}_{\mathrm{p}}$ be a subset of $\mathscr{L}$ such that labels in $\mathscr{L}_{\mathrm{p}}$ are used in tagging program points. Labels in $\mathscr{L} \setminus \mathscr{L}_{\mathrm{p}}$ are used in tagging values.

### 4.2.2 Interfaces

Each program point $p \in \mathscr{L}_{\mathrm{p}}$ is endowed with a subset of variables $I(p) \subseteq \mathscr{V}$. The set $I(p)$ is called the interface of the syntactic component $p$. The interface $I(p)$ is the set of the variables that are used by each thread at program point $p$.

### 4.2.3 Partial interactions

The behavior of a thread instance not only depends on its interface (which is dynamic), but also on a syntactic piece of code which describes how the thread instance uses its interface. This information is static because it is the same for each instance of a same thread. Each program point is endowed with a set of partial interactions. A partial interaction is given by a tuple. The first component is a partial interaction name $s$. The second and the third components are two sequences of variables. They allow for relating the formal variables of the rule to the syntactic variables of the threads. The first variable sequence (i.e. the second component) denotes the sequence of the variables that are associated with the values that are useful to check synchronization constraints and to compute values that are passed to other threads (in the case of the $\pi$-calculus, it is the variable that is bound to the name of the channel on which is sent a message and the variables that are bound to the channel names which compose this message). The second sequence (i.e. the third component) encodes the sequence of the variables that will be bound by some value passed by other threads during the interaction. The next component is a set of local constraints. It allows for some positive and negative matching among the values that are bound to the variables. The last component describes a set of potential continuations. We use a set of continuations to allow non-deterministic choices among several potential continuations. Each continuation is a set of threads. Each of these threads is defined by a program point, a marker, and an environment. This marker and this environment can only be known at run-time. Moreover, they are different for each instance of the same thread. Nevertheless, some information remains static: for instance, in the case of the $\pi$-calculus, the labels of the restrictions which declare fresh channel names are the same for every instances of the same thread instance. This static information is summarized inside a partial static environment $E_s$ that maps a subpart of the launched thread interface into static labels in $\mathscr{L}$.

**Example 4.2.1.** *In the case of the $\pi$-calculus, let $Q$ be a program point tagged with the label $q \in \mathscr{L}_p$. A thread at program point $[t = u]a?[x;y](\nu\ z)Q$ will be associated with the partial interaction defined by:*

- *the partial interaction name $in_2$, which means that this thread waits for the emission of two names over a channel;*
- *the parameter sequence $[a]$ which means that the thread has to exhibit the name that is associated with the variable $a$ to check whether the thread which it interacts with sends a message on the right channel;*
- *the sequence $[x,y]$ of bound variables means that the two variables $x$ and $y$ are bound to some values that are passed by some other threads during the interaction;*
- *the local constraint set $\{t = u\}$ means that the thread may actually exhibit the capability to compute the reception only if the values that are bound to the variables $t$ and $u$ are the same;*

- *the continuation set:*
$$\{\{(q,[z \to z])\}\},$$
*which means that there is only one continuation which consists in launching an instance of the syntactic component Q. Moreover, the variable z is bound to the name* $(z, id)$ *(where id is the marker of the computing thread). The values that are linked to the other variables (i.e. especially those obtained by name passing) will be computed by using the global interaction rule.*

This yields the following definition:

**Definition 4.2.2 (partial interaction).** A partial interaction $pi$ is given by a tuple $(s,(parameter_i),(bound_i),constraints,continuation)$, where

- $s$ is a partial interaction name in $\mathscr{A}$.

  We denote by $arity(s) = (m,n)$ its arities.
- $(parameter_i) \in \mathscr{V}^m$ is a finite sequence of variables,
- $(bound_i) \in \mathscr{V}^n$ is a finite sequences of distinct variables,
- the set $constraints \subseteq \{v \diamond v' \mid (v,v') \in \mathscr{V}^2,\ \diamond \in \{=;\neq\}\}$ describes some synchronization constraints,
- the set $continuation \in \wp(\wp(\mathscr{L}_{\mathrm{p}} \times (\mathscr{V} \rightharpoonup \mathscr{L})))$ is a set of potential continuations. We use a set to allow non-determinism. Each *potential continuation* is a set of continuation threads. A *continuation thread* is described by a pair $(q,E_s)$ where $q$ is a program point label and $E_s$ is a partial function from $\mathscr{V}$ into $\mathscr{L}$.

### 4.2.4 Initial states

The set of the initial states[1] is described by a set of potential continuations in $\wp(\wp(\mathscr{L}_{\mathrm{p}} \times (\mathscr{V} \rightharpoonup \mathscr{L})))$.

### 4.2.5 System syntax

The syntax of a system is given by the interface of each program point, the set of partial interactions of each program point, and the description of the initial states. We also require some compatibility properties. When the computation of a thread at program point $p$ launches a computation thread at program point $q$, all variables in the interface of the launched thread must be defined. There are

[1]We have several initial states, because we start by a non-deterministic choice

three cases: they may be already defined at program point $p$; they may be bound during the interaction; or they may be in the domain of the static environment of the continuation thread. Moreover, initial threads must define all variables in their interfaces (as being in the domain of their static environment). On the other hand, the partial interaction parameters and the variables that are used in matching must be in the interface of the threads that compute them. This yields the following definition for the syntax of a mobile system.

**Definition 4.2.3 (system syntax).** The syntax $syntax(\mathscr{S})$ of a mobile system $\mathscr{S}$ is a triple $(I, init_s, interaction)$ where:

- $I \in \mathscr{L}_{\mathrm{p}} \to \wp(\mathscr{V})$ maps each program point $p$ into the interface of the threads at the program point $p$;
- $init_s \in \wp(\wp(\mathscr{L}_{\mathrm{p}} \times (\mathscr{V} \rightharpoonup \mathscr{L})))$ is the description of the initial states;
- *interaction* maps each program point $p$ to the set of partial interactions that may be computed at program point $p$.

such that the three following conditions are satisfied:

1. For each program point label $p$ in $\mathscr{L}_{\mathrm{p}}$, for each partial interaction $(s, (parameter_i), (bound_i), constraints, continuation)$ in $interaction(p)$, and for each continuation thread $(q, E_s)$ in $\bigcup continuation$, the set $I(q)$ is included in the set $I(p) \cup \{bound_i\} \cup Dom(E_s)$;
2. For each $(q, E_s) \in \bigcup init_s$, the interface $I(q)$ is included in the domain $Dom(E_s)$;
3. For each program point $p$ in $\mathscr{L}_{\mathrm{p}}$ and for each partial interaction $(s, (parameter_i), (bound_i), constraints, continuation)$ in $interaction(p)$, we have $\{parameter_i\} \subseteq I(p)$ and $\{x; y\} \subseteq I(p)$ for any $(x \diamond y) \in constraints$.

## 4.3 System configurations

Each instance of threads and of values is stamped with a marker which encodes the history of the replication which have led to the creation of the thread instance, or to the value declaration. These markers are trees: we introduce $\mathscr{M}$ as the set of all trees, where each node is labeled with a sequence of program point labels in $\mathscr{L}_{\mathrm{p}}$ and where leaves are labeled with the symbol $\varepsilon$ and where the length of the label of a node is always equal to the number of children of this node. A value is given by a couple $(l, id)$ where $l$ is a label in $\mathscr{L}$ and $id$ is a marker in $\mathscr{M}$. When we have $l \in \mathscr{L}_{\mathrm{p}}$, the value is a pointer to a thread, otherwise the value refers to a name. Let $V \subseteq \mathscr{V}$ be a set of variables, an environment over $V$ maps each variable $x \in V$ to a dynamic value. This yields the following definition:

**Definition 4.3.1 (environment).** An environment is a partial function $E$ from $\mathscr{V}$ to $(\mathscr{L} \times \mathscr{M})$. If $V \subset \mathscr{V}$ is a subset of variables, then an environment $E$ is called an environment over $V$ if and only if $Dom(E) = V$.

During system computation, a thread can be described by a program point $p$, an instance marker $id$, and an environment $E$ over the interface $I(p)$ which associates the variables in $I(p)$ with some dynamic values. It is worth noting that variables should be classified between the variables that are associated with some pointer values (such as thread locations in the case of the mobile ambients) and the variables that are associated with some name values. We omit these kind of details for the sake of clarity.

Then, we can give the definition of a thread instance:

**Definition 4.3.2 (thread).** A thread is given by a triple $(p, id, E_p)$, where $p \in \mathscr{L}_{\mathrm{p}}$ is a program point label, $id \in \mathscr{M}$ is an instance marker, and $E_p$ is an environment over the interface $I(p)$.

During system computation, the system configuration is given by a set of threads:

**Definition 4.3.3 (configuration).** A system configuration is a set of threads.

## 4.4 Operational semantics

We describe in this section the operational semantics of the languages that are described in our meta language.

### 4.4.1 Primitives

#### 4.4.1.1 Exhibited action

We say that a thread $t$ exhibits a partial interaction $pi$ if it is able to compute this partial interaction provided that the context (i.e. the other threads in the configuration) may compute the some complementary partial interactions of the partial interaction $pi$. More precisely, the thread $t$ exhibits the partial interaction $pi$ if and only if the partial interaction is available in $t$ and if the local synchronization constraints are satisfied by the environment values. This yields the following formal definition:

**Definition 4.4.1 (exhibited action).** Let $t = (p, id, E)$ be a thread and $pi = (s, (parameter_i), (bound_i), constraints, continuation)$ be a partial interaction. We say that the thread $t$ exhibits the partial interaction $pi$ and we write $exhibit(t, pi)$ if and only if the both following properties are satisfied:

1. $pi \in interaction(p)$,
2. $\forall (a \diamond b) \in constraints$, we have $E(a) \diamond E(b)$ ($\diamond \in \{=, \neq\}$).

#### 4.4.1.2 Global synchronizations

Global synchronizations relate the interface of the interacting threads. We now define a predicate *sync* to compute whether some tuples of threads may synchronize their computation according to a formal rule. This predicate is a relation among a tuple of threads, the sequence of their respective parameter sequences, and the generic synchronization conditions that are described in the rule.

**Definition 4.4.2 (global synchronization).** Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \le k \le n} = (p^k, id^k, E^k)_{1 \le k \le n}$ be an $n$-tuple of threads, and $(param_l^k)_{k,l}$ be an $n$-tuple of parameter sequences ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread) and *compatibility* be a set of synchronization constraints. The relation *sync* is defined as follows:

$$sync((t^k), (param_l^k)_{k,l}, compatibility) \stackrel{\Delta}{=} \forall a, b \in compatibility,\ \sigma(a) = \sigma(b),$$

where $\sigma : \begin{cases} X_l^k & \mapsto E^k(param_l^k) \\ I^k & \mapsto (p^k, id^k). \end{cases}$

It is worth noting that the assignment $\sigma$ associates each formal variable with the run-time value of the syntactic variable that it is encoded by this formal variable.

#### 4.4.1.3 Marker computation

When a thread computes a partial interaction, a marker is computed and passed to the continuation. If the partial interaction is not a replication, the marker is the one of the computed thread, otherwise it is made of both the labels and the markers of the computed threads. The primitive *marker* takes the type of a partial interaction name $t$, a tuple of threads and a thread index $k$ and computes the marker that is associated with the continuation of the $k$-th thread when it computes a partial interaction the name of which is of type $t$. This yields the following definition for the allocated marker.

**Definition 4.4.3 (computed marker).** Let $n \in \mathbb{N}$ be an integer. Let $t$ be a partial interaction type in $\{replication; computation; migration\}$. Let $(p^i, id^i, E^i)_{1 \le i \le n}$ be an $n$-tuple of threads. Let $k$ be an integer such that $1 \le k \le n$. The marker $marker(t, (p^i, id^i, E^i)_{1 \le i \le n}, k)$ is defined as follows:

$$marker(t, \left(p^i, id^i, E^i\right)_{1 \le i \le n}, k) \triangleq \begin{cases} N((p^1, \ldots, p^n), id^1, \ldots, id^n) & \text{if t} = replication \\ id^k & \text{otherwise.} \end{cases}$$

### 4.4.1.4 Removing threads

When computing an interaction, the computing threads that do not replicate themselves are removed. The primitive *remove*, which is defined in the following, gives the set of threads that must be removed when computing an interaction. Its argument is a set of pairs made of a thread and the type of the partial interaction that is computed by this thread.

**Definition 4.4.4.** Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \le k \le n}$ be an $n$-tuple of threads. Let $(\text{type}^k)_{1 \le k \le n}$ be an $n$-tuple of partial interaction types. The primitive *remove* is defined as follows:

$$remove\left(\left(t^k, \text{type}^k\right)_{1 \le k \le n}\right) \triangleq \{t_k \mid 1 \le k \le n,\ \text{type}^k \neq replication\}.$$

### 4.4.1.5 Shared environment

We now introduce a primitive *vpassing* to compute value passing. This primitive takes a thread index $i$, a tuple of threads, the sequence of the variables that are bound in the $i$-th thread during the interaction, the tuple of the parameter sequences of each thread, and the formal description of value passing. It returns the environment of the $i$-th thread after value passing.

**Definition 4.4.5 (value passing).** Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \le k \le n} = (p^k, id^k, E^k)_{1 \le k \le n}$ be an $n$-tuple of threads. Let $(bd_l)_l$ be a sequence of variables ($bd_l$ is associated with the $l$-th variable that is bound by the interaction in the $i$-th thread). Let $(param_l^k)_{k,l}$ be an $n$-tuple of parameter sequences ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread). Let *communications* be a partial map from $\mathcal{V}_f^Y$ into $\mathcal{V}_f^X \cup \mathcal{V}_f^I$. We define the environment $vpassing(i, (t^k), (bd_l), (param_l^k), communications)$ by:

$$E^i[bd_j \mapsto \sigma(communications(Y_j^i))],$$

where $\sigma = \begin{cases} X_l^k \mapsto E^k(param_l^k) \\ I^k \mapsto (p^k, id^k). \end{cases}$

#### 4.4.1.6 Launching a continuation

When a thread computes a partial interaction, it is removed and replaced with some other threads. This set of new threads is called a continuation. The marker of each thread in the continuation is either the marker of the computed thread or a fresh marker that is computed from the markers of all the interacting threads. The values associated with variables may have three distinct origins. Some variables were already in the interface of the computed thread: they keep the same value. Some variables are associated with some values that are passed by the other threads: these associations are computed by the primitive *vpassing*. Some variables are created just after the interaction and associated with a fresh value obtained by tagging a static label with the marker of the thread. We first introduce an auxiliary primitive *update* which allows for updating an environment with the binding of the new variables. This primitive takes the marker of the computed thread, the environment of the computed thread, and the static environment that is associated with the continuation thread. It returns the updated environment, which is obtained by associating fresh variables with the correct syntactic labels stamped with the marker of the computed thread.

**Definition 4.4.6 (environment updating).** Let *id* be a marker in $\mathscr{M}$. Let $E_d$ be an environment over $V_d$ (i.e. $E_d \in V_d \to \mathscr{L} \times \mathscr{M}$). Let $E_s$ be a static environment over $V_s$ (i.e. $E_s \in V_s \to \mathscr{L}$). We define the environment $update(id, E_d, E_s)$ over $V_d \cup V_s$ as follows:

$$update(id,E_d,E_s)(x) \triangleq \begin{cases} (E_s(x), id) & \text{if } x \in V_s \\ E_d(x) & \text{if } x \in V_d \setminus V_s. \end{cases}$$

We can now introduce the primitive *launch*, which describes the launching of the continuation of a partial interaction. This primitive requires the syntactic continuation of the partial interaction, the marker of the computed thread and its environment. It computes the set of all threads launched when computing the partial interaction.

**Definition 4.4.7 (continuation launching).** Let $Ct \in \wp(\mathscr{L}_{\mathrm{p}} \times (\mathscr{V} \rightharpoonup \mathscr{L}))$ be a continuation. Let $id \in \mathscr{M}$ be a thread marker. Let $E \in \mathscr{V} \rightharpoonup \mathscr{L} \times \mathscr{M}$ be an environment. We define the set $launch(Ct, id, E)$ of threads as follows:

$$launch(Ct,id,E) \triangleq \left\{ \left(t, id, (update(id,E,E_s))_{|I(t)}\right) \;\middle|\; (t,E_s) \in Ct \right\}.$$

#### 4.4.1.7 Broadcast value passing

Broadcast value passing consists in substituting all occurrences of a value in the system by another one. In practice, it allows pointer re-addressing. We first compute a substitution $\tau$. This substitution replaces each value that may be associated at run-time with some formal variables with one of these formal variables. In case several formal parameters are associated with the same value, we non-deterministically choose which formal variable replaces the value. We introduce a primitive *subs_choice* that computes the choice of all the substitutions. This primitive takes the tuple of the interacting threads, a binding between formal variables and the parameters of the threads that interact, and a substitution among formal variables. It computes a set of substitutions over the set $\mathscr{L} \times \mathscr{M}$ of values.

**Definition 4.4.8 (substitution choice).** Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \leq k \leq n} = (p^k, id^k, E^k)$ be an $n$-tuple of threads. Let $(param_l^k)_{k,l}$ be an $n$-tuple of parameter sequences ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread) and *broadcast* be a partial map from $\mathscr{V}_{\mathscr{R}}^I$ into $\mathscr{V}_{\mathscr{R}}^X \cup \mathscr{V}_{\mathscr{R}}^I$.

The set $\mathscr{D}$ of the values that are replaced by the substitution is given as follows:

$$\mathscr{D} \triangleq \{(p^k, id^k) \mid \exists k,\ I^k \in Dom(broadcast)\}.$$

Then, we define the set $subs\_choice((t^k), (parameter_l^k), broadcast)$ of potential substitutions as the set of mappings $\tau \in \mathscr{L} \times \mathscr{M} \rightarrow \mathscr{L} \times \mathscr{M}$ such that:

1. $\forall x \in (\mathscr{L} \times \mathscr{M}) \setminus \mathscr{D},\ \tau(x) = x$;
2. $\forall x \in \mathscr{D},\ \tau(x) \in \{\sigma(broadcast(I^k)) \mid x = (p^k, id^k)\}$,

   where $\sigma : \begin{cases} I^{k'} \mapsto (p^{k'}, id^{k'}) \\ X_{l'}^{k'} \mapsto E^{k'}(param_{l'}^{k'}). \end{cases}$

This substitution is simultaneously applied on all the threads, which yields the following definition:

**Definition 4.4.9 (broadcast value passing).** Let $\tau \in (\mathscr{L} \times \mathscr{M}) \rightarrow (\mathscr{L} \times \mathscr{M})$ be a substitution. Let $C$ be a set of threads. We now define $subs(\tau, C)$ by:

$$subs(\tau, C) \triangleq \{(q, id, \tau \circ E) \mid (q, id, E) \in C\}.$$

*Remark* 4.4.10. We consider a set of potential substitutions because we make no assumption about the model that we encode. When the model satisfies the marker freshness sufficient conditions that are given in Def.4.5.11 on page 66, distinct threads have distinct identities, so for any distinct integer $k$ and $l$, the formal variables $I^k$ and $I^l$ may not be associated with the same run-time value. This way, only one substitution may apply.

### 4.4.2 Transition system

We use these primitives in order to describe both initial states and the semantics of computation steps according to a formal rule. Initial states are obtained by launching a continuation in $init_s$ with an empty marker and an empty environment. Thus the set $\mathscr{C}_0$ of initial states is defined by:

$$\mathscr{C}_0 \overset{\Delta}{=} \{launch(continuation, \varepsilon, \emptyset) \mid continuation \in init_s\}.$$

Computation steps are described by a reduction relation in Fig. 4.1. We recall the different steps of this computation, as follows:

- *interaction enabling*:
  - first, we find some threads that exhibit the right partial interactions;
  - then, we check that their interfaces are compatible with the synchronization constraints of the formal rule;
- *interaction computation*:
  - we remove the threads that do not compute a replication;
  - we choose a syntactic continuation for each thread;
  - we compute dynamic data for each of these continuations:
    * we compute the marker;
    * we take into account name passing;
    * we create fresh variables and associate them with the right values;
    * we restrict the environment to the new interface;
  - we apply the broadcast substitution to the whole system in order to model potential re-addressing.

Each computation step is labeled with all the information we need to know the system updating. More precisely, a computation step label is of the form $(\mathscr{R}, ((t^1, pi^1, Ct^1), \ldots, (t^n, pi^n, Ct^n)), \tau)$, where $\mathscr{R}$ is a reduction rule which expects $n$ interacting threads; where for any $k \in [\![1;n]\!]$, the $k$-th thread that is involved in the interaction is the thread $t^k$, this thread computes the partial interaction $pi^k$ before launching the continuation $Ct^k$; and where $\tau$ is the broadcast substitution that is applied to any thread at the end of the computation step.

## 4.5 Marker and value freshness

In this section, we give some sufficient conditions that ensure that our allocation scheme always provides fresh values.

Let $C$ be a configuration.
Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule.
We suppose that there exist both:

- an $n$-tuple $(t^k)_{1\leq k\leq n} = (p^k, id^k, E^k)_{1\leq k\leq n} \in C^n$ of distinct threads
- and an $n$-tuple

  $$(pi^k)_{1\leq k\leq n} = (s^k, (parameter^k_l)_l, (bd^k_l)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$$

  of partial interactions,

such that:

1. $\forall k \in [\![1;n]\!],\ exhibit(t^k, pi^k)$;
2. $\forall k \in [\![1;n]\!],\ components(k) = s^k$;
3. $sync((t^1,\ldots,t^n), (parameter^k_l)_{k,l}, compatibility)$ is satisfied.

Then:

$$C \overset{(\mathscr{R},(\alpha^1,\ldots,\alpha^n),\tau)}{\longrightarrow} subs(\tau, C \setminus removed\_threads \cup new\_threads)$$

where:

- $\tau \in subs\_choice\left(\left(t^k\right)_k, \left(parameter^k_l\right)_{k,l}, broadcast\right)$;
- $removed\_threads = remove\left(\left(t^k, type(s^k)\right)_{1\leq k\leq n}\right)$;
- $new\_threads = \bigcup\{launch\left(Ct^k, \overline{id}^k, \overline{E}^k\right) \mid 1 \leq k \leq n\}$,

  with $\forall k \in [\![1;n]\!]$:

  - $Ct_k \in continuation^k$,
  - $\overline{id}^k = marker\left(type(s_k), \left(p^{k'}, id^{k'}, E^{k'}\right)_{1\leq k'\leq n}, k\right)$,
  - $\overline{E}^k = vpassing(k, (t^{k'})_{k'}, (bd^k_l)_l, (parameter^{k'}_l)_{k',l}, communications)$;
- $\forall k \in [\![1;n;]\!],\ \alpha_k = (t^k, pi^k, Ct_k)$.

Figure 4.1: Generic transition rule.

### 4.5.1 Marker freshness sufficient conditions

We describe some assumptions over system syntaxes. These assumptions help in proving that there is ambiguity neither between the markers associated with distinct instances of threads (see Sect. 4.5.2), nor between the markers associated with fresh values (see Sect. 4.5.3), when computing the operational semantics (see Sect. 4.4).

#### 4.5.1.1 Migrating threads

The threads that migrate may create neither new thread, nor new value. However, a migration may change the environment and the control point of a thread. Changing the control point allows the encoding of thread polarity: a cycle of migrations among two program points may describe a membrane that have a different behavior according to its orientation (as in the projective brane calculus [28]). Thus, we require that the continuations of migrating threads contain at most one thread (no thread creation) and that their static environment is empty (no value creation). This yields the following well-formedness condition over migrating threads:

**Definition 4.5.1 (well-formed migrating threads).** The interaction map *interaction* well-defines migrating threads if ans only if the following property is satisfied:

$$\forall p \in \mathscr{L}_{\mathrm{p}}, (s,(parameter_i),(bound_i),constraints,continuation) \in interaction(p),$$
$$\text{if } type(s) = migration, \text{then } \forall Ct \in continuation \setminus \{\emptyset\}, Ct \text{ matches } \{(q,\emptyset)\}.$$

#### 4.5.1.2 Program point classes

Migration may just change the environment or the polarity of a thread. To observe a thread progress, we consider the set of the program point labels up to a binary equivalence relation $\sim$. This equivalence relation $\sim$ is the strongest equivalence relation that relates the program point labels that are reachable through migration steps. We first define a relation $\leftrightsquigarrow$ that relates two labels such that one of these label is the label of a thread in the continuation of a partial interaction that may be computed at the program point labeled by the other label.

**Definition 4.5.2 (migration transition).** $p \leftrightsquigarrow q$ if and only if:

1. either there exists $(s,(parameter_i),(bound_i),constraints,continuation) \in interaction(p)$, such that both $type(s) = migration$ and $\{(q,\emptyset)\} \in continuation$,

2. or there exists $(s,(parameter_i),(bound_i),constraints,continuation) \in interaction(q)$, such that both $type(s) = migration$ and $\{(p,\emptyset)\} \in continuation$.

Then, we define the equivalence relation $\sim$ as the strongest equivalence relation that is compatible with the relation $\leftrightsquigarrow$.

**Definition 4.5.3 (migration equivalence relation).** We define $\sim \in \mathscr{L}_{\mathrm{p}}^2$ as the reflexive, symmetric, and transitive closure of the relation $\leftrightsquigarrow$. Thus, the relation $\sim$ is an equivalence relation. For any program point $p \in \mathscr{L}_{\mathrm{p}}$, we denote by $[p]_\sim$ its equivalence class (i.e. $[p]_\sim = \{q \in \mathscr{L}_{\mathrm{p}} \mid p \sim q\}$). We denote by $\mathrm{C}_\sim$ the set $\{[p]_\sim \mid p \in \mathscr{L}_{\mathrm{p}}\}$ of all the equivalence classes.

#### 4.5.1.3 Thread creation injectivity

First, we ensure that both at the beginning of the system and when computing a partial interaction at most one thread is launched in each program point class.

**Definition 4.5.4 (thread creation injectivity).** The syntax $(I, init_s, interaction)$ creates threads injectively if and only if the following properties are satisfied:

1. for any $Ct \in init_s$, for any $(q, E_s), (q', E_s') \in Ct$, $q \sim q' \Rightarrow (q = q'$ and $E_s = E_s')$;
2. for any $p \in \mathscr{L}_{\mathrm{p}}$, for any $(s, (parameter_i), (bound_i), constraints, continuation) \in interaction(p)$, for any $Ct \in continuation(p)$, for any $(q, E_s), (q', E_s') \in Ct$, $q \sim q' \Rightarrow (q = q'$ and $E_s = E_s')$.

#### 4.5.1.4 Syntactic forest

In practice, a mobile system is described by a syntax tree (or a syntactic forest). This assumption is crucial to ensure the freshness of the markers that are associated with thread instances, since it prevents cyclic computation sequences. That is why we restore the notion of syntactic forest.

We first define the set of the successors of a program point class.

**Definition 4.5.5 (syntactic successor).** We say that the class $\mathrm{C}_2$ is a syntactic successor of the class $\mathrm{C}_1$ (and we write $\mathrm{C}_1 \longrightarrow_s \mathrm{C}_2$) if and only if $\mathrm{C}_1$ contains a program point label $p$ and $\mathrm{C}_2$ contains a program point label $q$ such that $q$ is the label of a continuation thread of a partial interaction in $interaction(p)$ that is not of type *migration*. This way, we define $\mathrm{C}_1 \longrightarrow_s \mathrm{C}_2$ by:

$$\begin{cases} \exists p \in \mathrm{C}_1, q \in \mathrm{C}_2, \\ \exists (s, (parameter_i), (bound_i), constraints, continuation) \in interaction(p), \\ \text{such that } \exists E_s \in \mathscr{V} \rightharpoonup \mathscr{L},\ (q, E_s) \in \bigcup continuation \text{ and } type(s) \neq migration. \end{cases}$$

A program point class is initial if and only if it is the successor of no other program point class:

**Definition 4.5.6 (initial program point classes).** A program point class $\mathrm{C}_2$ is initial if and only if:

$$\forall \mathrm{C}_1 \in \mathrm{C}_\sim,\ \mathrm{C}_1 \not\longrightarrow_s \mathrm{C}_2.$$

The set of all initial program point classes is denoted by $init_p$.

At the beginning of the computation, any thread in the system is an instance of of threads at initial program points. This yields the following well-formedness condition over the description of the initial states:

**Definition 4.5.7 (well-formedness initial state description).** The description of the initial states $init_s$ is well-formed if and only if for any $(p, E_s)$ in $\bigcup init_s$, we have $[p]_\sim \in init_p$.

The syntax of a system must define a syntax forest. So, for each program point class $\mathrm{C}_2$, there must be at most one edge in $\longrightarrow_s$ that ends in $\mathrm{C}_2$.[2] Moreover, we require that two partial interactions that compute threads at program points indexed with labels in the same class and that may launch threads at program points indexed with labels in the same class have the same type.

**Definition 4.5.8 (syntax forest).** The interaction map *interaction* of a program defines a syntax forest if and only if the following properties are satisfied:

1. $\forall \mathrm{C}_2 \in \mathrm{C}_\sim,\ \forall \mathrm{C}_1, \mathrm{C}'_1 \in \mathrm{C}_\sim, [\mathrm{C}_1 \longrightarrow_s \mathrm{C}_2 \text{ and } \mathrm{C}'_1 \longrightarrow_s \mathrm{C}_2] \implies \mathrm{C}_1 = \mathrm{C}'_2$.
2. for any $p, p' \in \mathscr{L}_\mathrm{p}$, for any $(s, \_, \_, \_, continuation) \in interaction(p)$, for any $(s', \_, \_, \_, continuation') \in interaction(p')$, for any $(q, E_s) \in \bigcup continuation$, for any $(q', E'_s) \in \bigcup continuation'$, if $p \sim p'$ and $q \sim q'$, then $type(s) = type(s')$.

### 4.5.1.5 Fresh values

When computing a partial interaction or at the beginning of the system computation, a thread declares some variables (excluding the variables that are bound by name passing) and associates them with some fresh values. We define here the set of the static labels of the fresh values that may be declared at a given program point class or at the beginning of the system computation.

**Definition 4.5.9.** The set $value(\mathrm{C}_1)$ of the values that may be declared when computing an instance of a thread at a program point in the class $\mathrm{C}_1$ is defined as follows: the set $value(\mathrm{C}_1)$ is the set of all the value $E_s(x)$ such that there exists a program point label $p$ in $\mathrm{C}_1$, a partial interaction

[2] We are only interested in the program point classes that are reachable from an initial program point class, so that we do not need to forbid cyclic paths.

$(s,(parameter_i),(bound_i),constraints,continuation) \in interaction(p)$, a continuation thread $(q,E_s) \in \bigcup continuation$ and a variable $x \in Dom(E_s) \cap I(q)$.

Moreover, the set $value(0)$ of the values that may be declared in an initial state is defined as follows:

$$value(0) \stackrel{\Delta}{=} \bigcup\{E_s(x) \mid (q,E_s) \in init_s,\ x \in Dom(E_s)\}.$$

In a syntactic forest, name restrictions are well-labeled if restriction symbols are associated with distinct labels. We translate this assumption into the syntax $syntax(\mathscr{S})$. More precisely, we require that the sets of the static labels of the fresh values that may be declared at a program point class are disjointed pairwise. This yields the following definition:

**Definition 4.5.10 (fresh values well-definition).** The syntax $syntax(\mathscr{S})$ well-defines its fresh values if and only if the following property is satisfied:

$$\forall \mathrm{C}_1,\mathrm{C}_2 \in \mathrm{C}_\sim \cup \{0\},\ [\mathrm{C}_1 \neq \mathrm{C}_2 \implies value(\mathrm{C}_1) \cap value(\mathrm{C}_2) = \emptyset].$$

#### 4.5.1.6 Sufficient conditions

We summarize our sufficient conditions in the following definition:

**Definition 4.5.11 (sufficient conditions).** The syntax $syntax(\mathscr{S}) = (I, init_s, interaction)$ allocates unambiguous markers if and only if the following properties are satisfied:

1. the map *interaction* well-defines migrating threads (Cf. Def. 4.5.1);
2. the syntax $syntax(\mathscr{S})$ creates threads injectively (Cf. Def. 4.5.4);
3. the description of the initial state $init_s$ is well-formed (Cf. Def. 4.5.7);
4. the interaction map *interaction* defines a syntactic forest (Cf. Def. 4.5.8);
5. the interaction map *interaction* well-defines fresh values (Cf. Def. 4.5.10).

In the rest of this section, we suppose that our systems satisfy Def. 4.5.11.

### 4.5.2 Thread marker freshness

Let $C_0 \in \mathscr{C}_0$ be an initial state and let $C_0 \xrightarrow{\lambda(1)} \ldots \xrightarrow{\lambda(n)} C_n$ be a computation sequence. For any $i$ such that $1 \leq i \leq o$, we denote $\lambda(i) = (\mathscr{R}(i),((t^1(i),pi^1(i),Ct^1(i)),\ldots,(t^{n_i}(i),pi^{n_i}(i),Ct^{n_i}(i))),\tau(i))$ where $\mathscr{R}(i)$ is a formal rule (we also denote $\mathscr{R}(i) =$

$(n(i), \mathit{components}(i), \mathit{compatibility}(i), \mathit{v\text{-}passing}(i), \mathit{broadcast}(i))$, $(t^k(i))$ is an $n(i)$-tuple of threads (we also denote $t^k(i) = (p^k(i), id^k(i), E^k(i))$), $(pi^k(i))$ is an $n(i)$-tuple of partial interactions (we also denote $(pi^k(i)) = (s^k(i), (\mathit{parameter}^k_l(i))_l, (bd^k_l(i))_l, \mathit{constraints}^k(i), \mathit{continuation}^k(i)))$, $(Ct^k(i))$ is an $n(i)$-tuple of continuations that satisfy $Ct^k(i) \in \mathit{continuation}^k(i)$ for any $k$ such that $1 \leq k \leq n(i)$, and $\tau(i)$ is a value substitution. For any $i \in [\![1;o]\!]$, we define the $n(i)$-tuple of markers $(\overline{id}^k(i))$ by $\mathit{marker}(\mathit{type}(s^k(i)), (t^{k'}(i))_{1\leq k'\leq n(i)}, k)$ and the $n(i)$-tuple of environments $(\overline{E}^k(i))$ by $\mathit{vpassing}(k, (t^{k'}(i))_{1\leq k'\leq n(i)}, (bd^k_l(i))_l, ((\mathit{parameter}^{k'}_l(i))_l), \mathit{communications}(i))$. We write $\mathit{consumed}(i,k)$ for the set of threads that are removed when computing the $k$-th thread in the $i$-th computation step and $\mathit{created}(i,k)$ for the set of threads that are launched when computing the $k$-th thread in the $i$-th computation step. They are defined as follows:

- $\mathit{consumed}(i,k) \triangleq \mathit{remove}\left(\left(t^k(i), s^k(i)\right)\right)$;
- $\mathit{created}(i,k) \triangleq \begin{cases} C_0 & \text{if } i = k = 0 \\ \mathit{launch}(Ct^k(i), \overline{id}^k(i), \overline{E}^k(i)) & \text{otherwise.} \end{cases}$

The set $\mathit{created}(0,0)$ denotes the initial threads. It is worth noting that we define the set of threads that are launched before applying the broadcast substitution, since this substitution only applies to thread environments (it modifies neither program point labels, nor thread markers).

The following lemma ensures that any thread in the system has been previously created either during a computation step or during a resource fetching, modulo some environment updating and some migration steps:

**Lemma 4.5.12.** *For any $i \in [\![0;o]\!]$ and for any thread $(p, id, E) \in C_i$, there exists a 4-tuple $(p', i', k, E')$ such that $i'$ and $k$ are two integers and $E'$ is an environment over $I(p')$ such that [ $i' = k = 0$ or ($1 \leq i' \leq i$ and $1 \leq k \leq n_{i'}$ and $\mathit{type}(s^k(i')) \neq \mathit{migration}$)] and such that both $p \sim p'$ and $(p', id, E') \in \mathit{created}(i', k)$.*

The freshness of allocated markers is expressed by Thm. 4.5.13 as follows:

**Theorem 4.5.13.** *Let $i$ be an integer between $0$ and $o$ and let $(p, id, E)$ be a thread in $C_i$. Then, there exists an unique 4-tuple $(p', i', k, E')$ such that $p'$ is a program point label, $i'$ and $k$ are integers, and $E'$ is an environment that satisfy $p \sim p'$, $(p', id, E') \in \mathit{created}(i', k)$ and such that either $i = 0$, or $\mathit{type}(s^k(i)) \neq \mathit{migration}$.*

So, for each thread, the class of the program point and the marker completely identify both the computation step where the thread has been created and which thread has been computed to launch it. Moreover, each partial interaction computation may launch at most one instance of a same thread. The proof of this theorem is given in Appendix B.1.

### 4.5.3 Fresh values

We now define the set of values that are used in a system at the $i$-th step of the computation:

**Definition 4.5.14 (used values).** We define the set *used_values*$(i)$ by:

$$\{E(x) \mid (p,id,E) \in C_i, x \in I(p)\}.$$

We now define the set *declared_value*$(i,k)$ of the values that have been declared during the $i$-th computation step by the $k$-th interacting thread, as follows:

**Definition 4.5.15 (declared values).** We define the set *declared_value*$(i,k)$ by:

$$\begin{cases} \mathit{used\_values}(0) & \text{if } k = i = 0, \\ \{(E_s(x), \overline{id}^k(i)) \mid (p,E_s) \in Ct^k(i),\ x \in Dom(E_s)\} & \text{otherwise .} \end{cases}$$

The set *declared_value*$(0,0)$ denotes the set of the initial values. The Lemma 4.5.16 ensures that any value that occurs in the system has been previously declared:

**Lemma 4.5.16.** *For any $i \in [\![0;o]\!]$, the set of value used_values$(i)$ is included in the set:*

$$\mathit{declared\_value}(0,0) \cup \bigcup \left\{ \mathit{declared\_value}(i',k) \,\middle|\, \begin{array}{l} 1 \le i' \le i, \\ 1 \le k \le n(i') \end{array} \right\}.$$

The Thm. 4.5.17 ensures that a given value may only be declared once in a system computation.

**Theorem 4.5.17 (value freshness).** *The following property is satisfied:*

$$\mathit{declared\_value}(i,k) \cap \mathit{declared\_value}(i',k') \neq \emptyset \implies k = k' \text{ and } i = i'.$$

*Proof.* Let $k$, $k'$, $i$ and $i'$ be four integers and $a$ be a value such that $a \in$ *declared_value*$(i,k)$ and $a \in$ *declared_value*$(i',k')$. We want to prove that $k = k'$ and $i = i'$.

By Def. 4.5.15, there exist two static environments $E_s$ and $E'_s$, two variables $x \in Dom(E_s)$ and $x' \in Dom(E'_s)$, and two program point classes $\mathrm{C}_1$ and $\mathrm{C}_2$ in $\mathrm{C}_\sim \cup \{0\}$ such that $a = (E_s(x), \overline{id}^k(i)) = (E'_s(x'), \overline{id}^{k'}(i'))$ (we set $\overline{id}^0(0) = \varepsilon$) and $E_s(x) \in value(\mathrm{C}_1)$ and $E'_s(x') \in value(\mathrm{C}_2)$. By Def. 4.5.10, we have $\mathrm{C}_1 = \mathrm{C}_2$. Then, there exist two environments $E$ and $E'$, and two program point label $p$ and $p'$ such that $p \sim p'$, $(p, \overline{id}^k(i), E) \in created(i,k)$, and $(p', \overline{id}^{k'}(i'), E') \in created(i',k')$. Since we have $\overline{id}^k(i) = \overline{id}^{k'}(i')$ and $p \sim p'$, we deduce from Thm. 4.5.13, that $E = E'$, $k = k'$ and $i = i'$. □

Moreover, according with the following proposition, we can simplify the shape[3] of the markers without losing marker allocation consistency:

**Proposition 4.5.18.** *Let $\phi_1$ and $\phi_2$ be the two following functions:*

$$\phi_1 = \begin{cases} \mathscr{M} & \rightarrow & (\mathscr{L}^*)^* \\ N((a_1,\dots,a_n),b_1,\dots,b_n) & \mapsto & \phi_1(b_2).(a_1,\dots,a_n)) \\ \varepsilon & \mapsto & \varepsilon, \end{cases}$$

$$\phi_2 = \begin{cases} \mathscr{M} & \rightarrow & \mathscr{L}^* \\ N((a_1,\dots,a_n),b_1,\dots,b_n) & \mapsto & \phi_2(b_2).a_2 \\ \varepsilon & \mapsto & \varepsilon. \end{cases}$$

*Marker allocation remains consistent when replacing each marker by its image by $\phi_1$ or $\phi_2$.*

*Proof.* Thanks to Def. 4.1.6, the proof of Thm. 4.5.13 is still valid. □

Such simplifications allow us to reduce the cost of our analysis, but also lead to a loss of accuracy, since they merge information related to distinct computation sequences of the system.

## 4.6 Conclusion

We have proposed a unifying framework to generically describe the semantics of mobile models. This framework encodes explicitly the relation between thread instances and what they create. It allows name matching, synchronization among several threads, value passing and broadcast substitution. In the next chapter, we will instantiate our framework in order to describe the semantics of the most popular mobile models. This way, we will show how to deal with channeled communication, migration, dissolution, encryption, decryption, and recursion...

[3]We have reversed markers in order to restore the chronological order (so that the past is on the left and the future in on the right)

# Chapter 5

# Encoding examples

In this chapter, we use the meta language that we have introduced in Chap. 4 in order to encode the most frequently used models for mobility in the current literature: We provide an encoding of a $\pi$-calculus version in Sect. 5.1, the *join*-calculus in Sect. 5.2, the *spi*-calculus in Sect. 5.3, the *ambients* in Sect. 5.4, and the BIO-*ambients* in Sect. 5.5.

This way, our meta language allows for the description of: internal choices (in Sect. 5.1) and external choices (in Sect. 5.1 and in Sect. 5.5); guarded replication (in Sect. 5.1, in Sect. 5.3, and in Sect. 5.4) and explicit recursion (in Sect. 5.2 and in Sect. 5.5); location, migration, and dissolution (in Sect. 5.4 and in Sect. 5.5); term construction and term destruction (in Sect. 5.3); safe migration (in Sect. 5.5) and channeled communication across boundaries (in Sect. 5.5).

Our meta language may also mix these features easily. Nevertheless we cannot model the bang operator, since spontaneous replication prevents from tracking the thread history. Moreover, we cannot deal with an equational theory such as in the applied-$\pi$ calculus [1], or with symmetric communications such as in the solo-calculus [49, 48] or in the fusion-calculus [65], since this feature may destroy the origin of the values that are used in the system. We have also avoided testing equalities among terms (in the *spi*-calculus we have assumed that we only match atomic names or public part of atomic keys). We do not show how we can deal with higher order communications [70] although our meta language may model them. We explain how we can extend our framework in Sect. 8.2.11. We also sketch the encoding of the projective brane calculus [28].

## 5.1 Revisiting the $\pi$-calculus

In this section, we consider a version of the synchronous polyadic $\pi$-calculus which handles with both internal and external choices, name matching, and guarded replication. We use the meta language that we have introduced in Chap. 4 to describe a non standard semantics for it.

## 5.1.1 A polyadic $\pi$-calculus with external choice

### 5.1.1.1 Syntax

Let $\mathscr{N}$ be a countable set of channel names. Let $\mathscr{L}_\mathrm{p}$ be a set of program point labels and $\mathscr{L}_\mathrm{n}$ be a set of name restriction labels. We suppose that $\mathscr{L}_\mathrm{p} \cap \mathscr{L}_\mathrm{n} = \emptyset$ and we denote by $\mathscr{L}$ the set of labels $\mathscr{L}_\mathrm{p} \cup \mathscr{L}_\mathrm{n}$. The syntax of threads is described in Fig. 5.1. Program points are external choices between some threads $\Sigma_i^l M_i.act_i.Q_i$: they are labeled by distinct labels $l \in \mathscr{L}_\mathrm{p}$. Input guard, replication guard, and name restriction act as name binders, i.e in the threads $M.c?[x_1,\dots,x_n].P$, $M.*d?[y_1,\dots,y_p].Q$ and $(\nu^\alpha x)R$, the occurrences of $x_1,\dots,x_n$ in $P$, $y_1,\dots,y_p$ in $Q$ and $x$ in $R$ are bound. Moreover, we suppose that in the thread $M.c?[x_1,\dots,x_n].P$ or in the thread $M.*c?[x_1,\dots,x_n].P$, the names $x_1,\dots,x_n$ are pairwise distinct. Name restrictions are labeled with distinct labels in $\mathscr{L}_\mathrm{n}$. Usual rules about scope, substitution, and $\alpha$-conversion apply. We denote by $fn(P)$ the set of the names that are free in $P$, i.e names that are not under the scope of a binder, and by $bn(P)$ the set of the names that are bound in $P$.

$$
\begin{array}{ccl}
x,y,a & \in & \mathscr{N} \\
\overline{x},\overline{y} & \in & \mathscr{N}^* \\
\alpha & \in & \mathscr{L}_\mathrm{n} \\
l & \in & \mathscr{L}_\mathrm{p} \\
I & \in & \{[\![1;n]\!] \mid n \in \mathbb{N}^*\} \\
 & & \\
P & ::= & \Sigma_{i\in I}^l M_i.act_i.P_i \mid (P \mid P) \mid (P \oplus P) \mid (\nu^\alpha x)P \mid \mathbf{0} \\
M & ::= & [x=y].M \mid [x \neq y].M \mid \varepsilon \\
act & ::= & a?[\overline{y}] \mid a![\overline{x}] \mid *a?[\overline{y}]
\end{array}
$$

Figure 5.1: Syntax.

An external choice $\Sigma_{i\in I}^l M_i.act_i.P_i$ between threads may compute the action $act_i$ and may launch the continuation $P_i$ only if on the left hand the matching conditions $M_i$ are satisfied and on the right hand its concurrent threads may perform the corresponding co-actions.

### 5.1.1.2 Semantics

As usual, the operational semantics is given by both a transition relation in Figs. 5.3 and 5.4, and a congruence relation in Fig. 5.5. The transition relation

$$\frac{}{\Sigma_i M_i.act_i.P_i \Rightarrow \Sigma_i M_i.act_i.P_i} \qquad \frac{}{\mathbf{0} \Rightarrow \mathbf{0}}$$

$$\frac{P_1 \Rightarrow Q}{P_1 \oplus P_2 \Rightarrow Q} \qquad \frac{P_2 \Rightarrow Q}{P_1 \oplus P_2 \Rightarrow Q}$$

$$\frac{P_1 \Rightarrow Q_1,\ P_2 \Rightarrow Q_2}{(P_1 \mid P_2) \Rightarrow (Q_1 \mid Q_2)} \qquad \frac{P \Rightarrow Q}{(\nu^{\alpha} x).P \Rightarrow (\nu^{\alpha} x).Q}$$

Figure 5.2: Continuation computation.

$$\frac{\models M_{i_0},\ \models M'_{j_0},\ act_{i_0} = a?[\overline{y}],\ act'_{j_0} = a![\overline{x}],\ P_{i_0}[\overline{y} \leftarrow \overline{x}] \Rightarrow Q,\ P'_{j_0} \Rightarrow Q'}{\Sigma^{l}_{i \in I} M_i.act_i.P_i \mid \Sigma^{l'}_{j \in J} M'_j.act'_j.P'_j \overset{com(l,l')}{\longrightarrow} Q \mid Q'}$$

$$\frac{\models M_{i_0},\ \models M'_{j_0},\ act_{i_0} = *a?[\overline{y}],\ act'_{j_0} = a![\overline{x}],\ P_{i_0}[\overline{y} \leftarrow \overline{x}] \Rightarrow Q,\ P_{j_0} \Rightarrow Q'}{\Sigma^{l}_{i \in I} M_i.act_i.P_i \mid \Sigma^{l'}_{j \in J} M'_j.act'{}_j.P'_j \overset{fetch(l,l')}{\longrightarrow} Q \mid Q' \mid \Sigma^{l}_{i \in I} M_i.act_i.P_i}$$

Figure 5.3: Interaction computation.

$$\frac{P \xrightarrow{\lambda} Q}{(\nu^{\alpha} x)P \xrightarrow{\lambda} (\nu^{\alpha} x)Q} \qquad \frac{P' \equiv P \quad P \xrightarrow{\lambda} Q \quad Q \equiv Q'}{P' \xrightarrow{\lambda} Q'} \qquad \frac{P \xrightarrow{\lambda} P'}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

Figure 5.4: Compatibility rules.

$$\begin{array}{rcll l}
(\nu^{\alpha} x)P & \equiv & (\nu^{\alpha} y)P[x \leftarrow y] & \text{if } y \notin fn(P) & (\alpha\text{-conversion}) \\
P \mid Q & \equiv & Q \mid P & & \text{(commutativity)} \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & & \text{(associativity)} \\
P \mid \mathbf{0} & \equiv & P & & \text{(end of a thread)} \\
(\nu^{\alpha} x)\mathbf{0} & \equiv & \mathbf{0} & & \text{(garbage collecting)} \\
(\nu^{\alpha} x)(\nu^{\beta} y)P & \equiv & (\nu^{\beta} y)(\nu^{\alpha} x)P & \text{if } x \neq y & \text{(swapping)} \\
((\nu^{\alpha} x)P) \mid Q & \equiv & (\nu^{\alpha} x)(P \mid Q) & \text{if } x \notin fn(Q) & \text{(extrusion)}
\end{array}$$

Figure 5.5: Structural congruence.

describes thread computation, whereas the congruence relation explicitly allows for thread interactions. An interaction is enabled when we have two concurrent threads such that the first one has a summand which may perform a reception on a channel and such that the second thread has a summand which may perform an emission over the same channel. Moreover the length of both the sent and the expected message must be the same, otherwise the interaction is disabled. The result consists in launching the corresponding two continuations, just after having applied name passing. We propose to directly launch a set of external choices among some partial interactions (internal choices are always systematically performed just after an interaction). The choice between the potential continuations is given by an auxiliary relation in Fig.5.2. More precisely, we write $P \Rightarrow Q$ if $Q$ is a system made of concurrent external choices such that $Q$ is obtained from $P$ by applying only internal choices. We also use an auxiliary relation $\models$ to check whether matching conditions are satisfied or not.

## 5.1.2 Non standard semantics

### 5.1.2.1 Partial interaction names

We distinguish communications according to the length of the communicated message. Thus, we define the set of partial interaction names, their types and their arities as follows:

$$\{out_n \mid n \in \mathbb{N}\} \cup \{in_n \mid n \in \mathbb{N}\} \cup \{fetch_n \mid n \in \mathbb{N}\};$$

$$Ari = \begin{cases} out_n & \mapsto (n+1, 0) \\ in_n & \mapsto (1, n) \\ fetch_n & \mapsto (1, n); \end{cases} \qquad type = \begin{cases} out_n & \mapsto computation \\ in_n & \mapsto computation \\ fetch_n & \mapsto replication. \end{cases}$$

The partial interaction name $out_n$ denotes a thread that is sending a message of $n$ names. Such a partial interaction is a regular computation, it has $n+1$ parameters: the first one is the name of the channel on which the message is sent and the others are the names of which the message consists. Such a partial interaction bounds no variable. The partial interaction name $in_n$ denotes a thread that is waiting for a message of $n$ names. Such a partial interaction is a regular computation. It has one parameter which is the name of the channel on which the message is waited for and it bounds $n$ variables. Finally, the partial interaction name $fetch_n$ denotes a thread that may replicate itself when receiving a message of $n$ names. Such a partial interaction is a replication. It has one parameter which is the channel on which the message is waited for and it bounds $n$ variables.

#### 5.1.2.2 Formal rules

We now define the formal rules that implement communication and resource fetching. The rule $com_n$ which is described in Fig. 5.6 implements the communication of $n$ names. It requires two threads: the first one waits for a message of $n$ names ($in_n$) and the second one sends a message of the same size ($out_n$). The synchronization condition $X_1^1 = X_1^2$ ensures that the first parameters of each component are associated with the same value, which means that the emission and the reception are performed on the same channel. At last, when interacting, the $i$-th bound variable of the receiver is associated with the value that is linked with the $(i+1)$-th parameter of the message sender which models name passing. There is no broadcast communication.

$$com_n = (2, component, synchronization, communication, global)$$

where

1. $component(1) = in_n$ and $component(2) = out_n$;
2. $synchronization = \{X_1^1 = X_1^2\}$;
3. $communication = [Y_i^1 \leftarrow X_{i+1}^2, \forall i \in [\![1;n]\!]\,]$;
4. $global = \emptyset$.

Figure 5.6: Formal rule for communication.

The rule $fetch_n$ which is described in Fig. 5.7 implements the replication of a resource when receiving a $n$ names message. It is exactly the same as the communication rule, except the fact that the second thread computes a replication $(fetch_n)$. The facts that the resource is still available after the computation step and that a new marker is associated with the new instance are dealt with systematically by the generic operational semantics (Cf. Sect. 4.4.2). That is why they are not described explicitly in the rule.

#### 5.1.2.3 Abstract syntax extraction

We now define the syntax extraction function that takes a program describing the initial state of a mobile system in the standard syntax and extracts the abstract syntax of it.

The interface of a program point $P$ labeled with $l$ is the set of the free names $fn(P)$ of $P$. The abstract syntax maps each program point label $l$ that is used to label the program point $\Sigma_{i\in I}^{l} M_i.a_i.P_i$ to the following set of partial interactions:

$$\{(act(a_i), arg(a_i), bound(a_i), flat(M_i), \beta(P_i, \emptyset)) \mid i \in I\},$$

$$fetch_n = (2, component, synchronization, communication, global)$$

where

1. $component(1) = fetch_n$ and $component(2) = out_n$;
2. $synchronization = \{X_1^1 = X_1^2\}$;
3. $communication = [Y_i^1 \leftarrow X_{1+i}^2,\ \forall i \in [\![1;n]\!]\ ]$;
4. $global = \emptyset$.

Figure 5.7: Formal rule for resource fetching.

where auxiliary primitives are defined as follows:

- The primitive *act* maps each action of the standard syntax into the corresponding partial interaction name in the abstract syntax; it is defined as follows:

$$act = \begin{cases} a?[y_1,\ldots,y_n] & \mapsto in_n \\ a![x_1,\ldots,x_n] & \mapsto out_n \\ *a?[y_1,\ldots,y_n] & \mapsto fetch_n. \end{cases}$$

- The primitive *arg* gives the set of the syntactic variables that are used as parameters of the partial interactions:

$$arg = \begin{cases} a?[y_1,\ldots,y_n] & \mapsto [a] \\ a![x_1,\ldots,x_n] & \mapsto [a;x_1;\ldots;x_n] \\ *a?[y_1,\ldots,y_n] & \mapsto [a]. \end{cases}$$

  In the case of a message reception and in the case of a replication, the only parameter is the variable that is bound at run-time to the name of the channel on which the reception is performed. In the case of a message emission, the first parameter is the variable that is bound at run-time to the name of the channel on which the reception is made, the others are the variables that are bound to the names that compose the message.

- The primitive *bound* maps each action of the standard syntax into the sequence of the variables that are bound when computing the partial interaction.

$$bound = \begin{cases} a?[y_1,\ldots,y_n] & \mapsto [y_1;\ldots;y_n] \\ a![x_1,\ldots,x_n] & \mapsto [] \\ *a?[y_1,\ldots,y_n] & \mapsto [y_1;\ldots;y_n]. \end{cases}$$

This sequence is empty in the case of a message emission, otherwise it is the sequence of the variables that are used as the arguments of the reception.

- The primitive *flat* computes inductively the set of synchronization constraints, as follows:

$$flat = \begin{cases} [x = y].M & \mapsto \{x = y\} \cup (flat(M)) \\ [x \neq y].M & \mapsto \{x \neq y\} \cup (flat(M)) \\ \varepsilon & \mapsto \emptyset. \end{cases}$$

- Finally the primitive $\beta$ computes the set of potential continuations. It is defined inductively over the standard syntax of the syntactic continuation, as follows:

$$\begin{aligned} \beta((\nu^{\alpha}x)P, E_s) &= \beta(P, E[x \mapsto \alpha]) \\ \beta(\mathbf{0}, E_s) &= \{\emptyset\} \\ \beta(P \oplus Q, E_s) &= \beta(P, E_s) \cup \beta(Q, E_s) \\ \beta(P \mid Q, E_s) &= \{A \cup B \mid A \in \beta(P, E_s),\ B \in \beta(Q, E_s)\} \\ \beta(\Sigma^{l}_{i \in J} a.P, E_s) &= \{\{(l, E_s)\}\}. \end{aligned}$$

The set of the initial states is defined as follows:

$$\beta(\mathscr{S}, \emptyset).$$

### 5.1.3 Correspondence

**Proposition 5.1.1.** *For each system, the extracted syntax satisfies the sufficient properties in Def. 4.5.11. Thus, allocated markers are unambiguous.*

The correspondence between the standard and the non standard semantics is established as follows:

**Theorem 5.1.2.** *The standard and the non standard semantics are in strong bisimulation.*

## 5.2 Encoding the *join*-calculus

The *join*-calculus [40] is a model for mobile systems that is based on the notion of locality. The $\pi$-calculus is based on the use of non-local interactions (such as *rendez-vous*) which are very difficult to implement. In the *join*-calculus, only the capability to send a message over a channel name may be passed to other threads. This way the messages that are communicated through a channel may only be received by exactly one thread in the system. This assumption allows for an easy and efficient distributed implementation of this model.

Locality is ensured by the use of recursive definitions. Recursive definitions open some fresh channels and launch continuations. Only the definition has the capability to receive messages over the channels it has opened. Nevertheless continuations may send messages on these channels and may communicate the capability to send messages on these channels to the other threads. Each definition may be fetched when there are simultaneous emissions over all channels it has opened.

For the sake of simplicity, we use the core version of the *join*-calculus that is described in [41, Sect. 5], where communications are monoadic and definitions always wait for two simultaneous outputs. The general case may be dealt with easily. Moreover, we separate the device (that is, the set of declared definitions) from the set of threads.

## 5.2.1 Syntax

Let $\mathscr{N}$ be a set of channel names. Let $\mathscr{L}_\mathrm{p}$ be a set of program point labels and $\mathscr{L}_\mathrm{n}$ be a set of name labels. We suppose that $\mathscr{L}_\mathrm{p} \cap \mathscr{L}_\mathrm{n} = \emptyset$ and we denote by $\mathscr{L}$ the set $\mathscr{L}_\mathrm{p} \cup \mathscr{L}_\mathrm{n}$. The syntax of threads is described in Fig. 5.8. Program points are

$$P ::= x^l \langle u \rangle \mid (P \mid P) \mid (\mathrm{def}^l\ x^\alpha \langle u \rangle \mid y^\beta \langle v \rangle \triangleright P \text{ in } P).$$
$$\text{where } u,x,y \in \mathscr{N},\ l \in \mathscr{L}_\mathrm{p} \text{ and } \alpha, \beta \in \mathscr{L}_\mathrm{n}.$$

Figure 5.8: Syntax for the *join*-calculus.

definitions and message outputs: A definition $\mathrm{def}^l\ x^\alpha \langle u \rangle \mid y^\beta \langle v \rangle \triangleright Q$ in $P$ defines a new resource. It opens two fresh channels named $x$ and $y$ in the continuation $P$ and in any recursive instance of the thread $Q$. Thus, the variables $x$ and $y$ are bound in $P$ and in $Q$. When receiving simultaneous messages over the channels $x$ and $y$, the definition may launch an instance of the thread $Q$ where the two variables $u$ and $v$ are associated with their respective messages. Thus, the variables $u$ and $v$ are bound in the thread $Q$ (the variables $u$ and $v$ must be distinct). A thread $x^m \langle u \rangle$ performs an asynchronous output on the channel name $x$. Usual rules about scope, substitution, and $\alpha$-conversion apply. We denote by $fn(P)$ the set of the variables that are free in $P$ and by $bn(P)$ the set of the variables that are bound in $P$. We assume that each label occurs only once in the whole system.

### 5.2.2 Semantics

The system state is given by a pair $(D,P)$ where $P$ is a system of threads and $D$ is a set of declared definitions of the form $x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright^{l} P$ that is called the device. We denote by $dn(D)$ the set of channel names $\bigcup\{\{x;y\} \mid (x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright^{l} P) \in D\}$. Each definition in $D$ may be fetched by threads of the system $P$.

As usual, the operational semantics is given by both a transition relation in Fig. 5.10 and a congruence relation in Fig. 5.9. The transition relation describes thread computation and definition storage, whereas the congruence relation explicitly allows for the thread interactions. A definition may be stored when it reaches the top level. In such a case, we rename the name it declares in order to avoid conflicts and stores it in the device. Such a definition is then available and may be fetched during the rest of the computation sequence. An interaction is enabled when we have a definition in the device and two concurrent outputs such that the definition waits for simultaneous messages on two channels $x$ and $y$ and such that the outputs are performed on the channels $x$ and $y$. The result consists in launching an instance of the definition body just after having applied name passing.

The congruence relation allows for the associativity and the commutativity of the parallel composition. It allows for renaming channels that are opened by each definition, so that they may be stored in the device.

$$\begin{array}{rcl}
\mathrm{def}^{l}\ x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright Q \ \mathrm{in}\ P & \equiv & \mathrm{def}^{l}\ z^{\alpha}\langle u\rangle \mid t^{\beta}\langle v\rangle \triangleright Q\sigma \ \mathrm{in}\ P\sigma \mathrm{if}\ z,t \notin fn(P \mid Q) \\
P \mid Q & \equiv & Q \mid P \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R \\
\multicolumn{3}{c}{\text{where } \sigma = [x \leftarrow z; y \leftarrow t].}
\end{array}$$

Figure 5.9: Congruence relation for the *join*-calculus.

$$\frac{x,y \notin dn(D)}{(D, \mathrm{def}^{l}\ x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright P \ \mathrm{in}\ Q) \xrightarrow{\varepsilon} (D \cup \{x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright^{l} P\}, Q)}$$

$$\frac{x^{\alpha}\langle u\rangle \mid y^{\beta}\langle v\rangle \triangleright^{l} P \in D}{(D, x^{j}\langle s\rangle \mid y^{k}\langle t\rangle) \xrightarrow{(i,j,k)} (D, P[u \leftarrow s; v \leftarrow t])}$$

$$\frac{(D,P) \xrightarrow{\lambda} (D',P')}{(D,P \mid R) \xrightarrow{\lambda} (D',P' \mid R)}$$

where $x, y, u, v \in \mathscr{N}$, $l \in \mathscr{L}_{\mathrm{p}}$, $\alpha, \beta \in \mathscr{L}_{\mathrm{n}}$.

Figure 5.10: Reduction relation for the *join*-calculus.

### 5.2.3 Non standard semantics

The non standard semantics is obtained by describing each state of the system by a set of thread instances where a thread may denote either a recursive definition or an output instance.

#### 5.2.3.1 Partial interaction names

We define the set of partial interaction names by:

$$\{def;\ output\};$$

Partial interaction type and arities are defined as follows:

$$\{def;\ output\};$$

$$Ari = \begin{cases} def & \mapsto (2,2) \\ output & \mapsto (2,0); \end{cases} \qquad type = \begin{cases} def & \mapsto replication \\ output & \mapsto computation \end{cases}$$

A partial interaction named *def* may be performed by a recursive definition. Such a partial interaction is a replication. It has two parameters that are the two names of the channels at which some messages are listened to. It bounds two variables to the channel names that will be received when fetching the definition. A partial interaction named *output* may be performed by a message output. This partial interaction is a regular computation. It has two parameters: the first one is the name of the channel on which the emission is made and the second one is the name that is sent. It bounds no variables.

#### 5.2.3.2 Formal rules

We now define the unique formal rule. The rule *fetch* which is described in Fig. 5.11 implements the definition fetching. It requires three threads: the first one is a definition (*def*) and the two others send a message (*output*).

The definition waits two simultaneous names over the channels the names of which are described by the formal variables $X_1^1$ and $X_2^1$. The first output is made over a channel the name of which is denoted by the formal variable $X_1^2$ and the second output is made over a channel the name of which is denoted by the formal variable $X_1^3$. Thus, the definition fetching is enabled if and only if the formal variables $X_1^1$ and $X_1^2$ encode the same channel name and the formal variables $X_2^1$ and $X_1^3$ encode the same channel name which is given by the compatibility condition set $\{X_1^1 = X_1^2; X_2^1 = X_1^3\}$. At last, two variables are associated with the names communicated by the output threads when launching an instance of the definition body. These two variables are encoded by the formal variables $Y_1^1$ and $Y_2^1$ and the two communicated names are encoded by the formal variables $X_2^2$ and $X_2^3$. This way name passing is described by the substitution $[Y_1^1 \leftarrow X_2^2, Y_2^1 \leftarrow X_2^3]$. There is no broadcast communication.

$$fetch = (3, component, synchronization, communication, global)$$

where:

1. $component = \begin{cases} 1 \mapsto def \\ 2 \mapsto output \\ 3 \mapsto output; \end{cases}$
2. $synchronization = \{X_1^1 = X_1^2; X_2^1 = X_1^3\}$;
3. $communication = [Y_1^1 \leftarrow X_2^2, Y_2^1 \leftarrow X_2^3]$;
4. $global = \emptyset$.

Figure 5.11: Formal rule for computation.

#### 5.2.3.3 Abstract syntax extraction

We now define the syntax extraction function that takes a program describing the initial state of a mobile system in the standard syntax and extracts its abstract syntax.

Program points are recursive definitions and message outputs. The abstract syntax maps each program point label $p \in \mathscr{L}_{\mathrm{p}}$ to the following set of partial interactions according to the syntax of the program point:

- the label of a program point $x^l\langle u\rangle$ is associated with the interface $\{x; u\}$ and the following set of partial interactions:

$$\{(output, [x;u], [], \emptyset, \{\emptyset\})\};$$

- the label of a program point $\mathrm{def}^l\ x^\alpha\langle u\rangle \mid y^\beta\langle v\rangle \triangleright Q$ in $P$ is associated with the interface $(fn(Q) \setminus \{u; v; x; y\})$ and the following set of partial interactions:

$$\{(def, [x;y], [u;v], \emptyset, \{\beta(Q, \emptyset)\})\}.$$

Continuation computations use an auxiliary primitive $\beta$. This primitive computes only one computation, since there is no non-deterministic choice. The primitive

$\beta$ is defined inductively over the standard syntax of the continuation, as follows:

$$\begin{array}{rcl} \beta(x^l\langle u\rangle, E_s) & = & \{(l, E_s)\} \\ \beta(P \mid Q, E_s) & = & \beta(P, E_s) \cup \beta(Q, E_s) \\ \beta(\mathrm{def}^l\ x^\alpha\langle u\rangle \mid y^\beta\langle v\rangle \triangleright Q \ \mathrm{in}\ P, E_s) & = & \{(l, E_s)\} \cup \beta(P, E_s[x \mapsto \alpha, y \mapsto \beta]). \end{array}$$

The set of initial states is defined as follows:

$$\beta(\mathscr{S}, \emptyset).$$

### 5.2.4 Correspondence

**Proposition 5.2.1.** *For each system, the extracted syntax satisfies the sufficient properties in Def. 4.5.11. Thus, allocated markers are unambiguous.*

The correspondence between the standard and the non standard semantics is established as follows:

**Theorem 5.2.2.** *The standard and the non standard semantics are in weak bisimulation.*

## 5.3 Encoding the *spi*-calculus

In this section we propose an encoding of the *spi*-calculus [3, 2]. The *spi*-calculus is a model of computation for describing cryptographic protocols. In this model, cryptographic primitives are ideal, which essentially means that the only way to decrypt an encrypted message is to have the correct key.

Encoding the *spi*-calculus addresses the problem of dealing with terms: the encryption of a message is represented by a term which describes the kind of encryption protocol, the encrypted message, and the key. Threads may decrypt such *cipher-text* in the case when they apply the correct decryption protocol with the correct key. Moreover, threads may pass some terms to each other by using channeled communications.

Here we consider a version of the *spi*-calculus with tuples, shared-key encryption, public-key encryption, and message signature. Shared-key encryption consists in sharing a key between two agents. The first one encrypts a message using the key, and anybody having this key has the capability to decrypt the encrypted message. In public-key encryption, each key has two parts. The public part is used to encrypt, but only agents who know the private part of the key may perform the decryption. Signature consists in associating the private part of a key to a message, so that any agent may check the signature with the public part of the key.

We suppose that shared keys and private parts of keys are atomic names. We also suppose that communication primitives are applied with channel names. This means that at run-time thread computation will be stopped in case of type-mismatch.

## 5.3.1 Syntax

Let $\mathscr{N}$ be a set of channel names and $\mathscr{V}$ be a set of variables. Let $\mathscr{L}_a$ be a set of thread labels, $\mathscr{L}_t$ be a set of term constructor labels, and $\mathscr{L}_\mathrm{n}$ be a set of name labels. We suppose that $\mathscr{L}_a$, $\mathscr{L}_t$ and $\mathscr{L}_\mathrm{n}$ are pairwise disjoint and we denote by $\mathscr{L}_\mathrm{p}$ the set $\mathscr{L}_a \cup \mathscr{L}_t \cup \mathscr{L}_\mathrm{n}$. The syntax of threads is described in Fig. 5.12.

A term may be a name, a variable, or a term constructor application. The term $n$ denotes a name which may describe a channel, a shared-key or the private part of a key. It has been previously introduced by a restriction $(\nu^\alpha n)$. The term $x$ is a variable which will be associated at run-time, either to a communicated term, or to the result of a term destruction. We use several term constructors and several term destructors. The term $\mathrm{pk}^t(M)$ denotes the public part of the private key $M$. At run-time, $M$ must be a name. The term $\mathrm{tuple}^t_n(M_1,\ldots,M_n)$ denotes an $n$-tuple of terms. The term $\mathrm{sencrypt}^t(M_1,M_2)$ denotes the result of a shared-key encryption of a message: the term $M_2$ denotes the shared key, the term $M_1$ is the message to encrypt. The term $\mathrm{pencrypt}^t(M_1,M_2)$ denotes the result of a public-key encryption: the term $M_2$ denotes the public part of the key and the term $M_1$ is the message to encrypt. The term $\mathrm{sign}(M_1,M_2)$ denotes the result of a message signature: the term $M_2$ denotes the private part of the key, and the term $M_1$ denotes the signed message.

Threads may declare new names, communicate some terms, and apply term destructors to terms. A thread $(\nu^\alpha n)P$ introduces a fresh variable and associates it with a fresh name $n$ in the continuation $P$. The thread $M^p\langle x\rangle.P$ waits for a term on a channel. When the communication is computed, the variable $x$ is declared in $P$ and associated with the received name. Conversely, the thread $\overline{M_1}^p\langle M_2\rangle.P$ sends a term over a channel. It is worth noting that any run-time type mismatch prevents further thread computation, that is to say that the term $M$ in $M^p\langle x\rangle.P$ (and the term $M_1$ in $\overline{M_1}^p\langle M_2\rangle.P$) must be bound with a name, otherwise no interaction may be performed. Events $\mathrm{begin}(M)$ and $\mathrm{end}(M)$ allow for expressing authentification specification. The term $M$ must be associated with some name at run-time. Proving the authentification of a protocol boils down to prove that an event $\mathrm{end}(n)$ may not reach the top level before its dual event $\mathrm{begin}(n)$ [7]. A thread $[M_1 = M_2]^p.P$ tests at run time that $M_1$ and $M_2$ are associated with some names or with the public part of atomic keys [1] and that these names or the two

[1]Normally, it tests equality without testing that $M_1$ and $M_2$ are names or key public part: we

atomic keys are the same. In such a case, the continuation $P$ is launched. A thread $(\mathrm{let}^{\alpha} x = M \text{ in } P)$ manipulates terms by using the appropriate term destructor. In case of success, it declares the variable $x$ in $P$ and associates it with the result of the term destruction. Destructors are $\mathrm{th}_i^n$ that extracts the $i$-th component of an $n$-tuple, sdecrypt that allows for the decryption of a shared-key encrypted message, pdecrypt that allows for the decryption of a public-key encryption by using the private part of the key, checksign that checks that the message is signed with the private version of a public key, and getmessage gives the message while ignoring the signature. Term destruction is blocked in case of type mismatch, i.e. an encryption with a shared-key which is not a name, or with a public part of a key the private part is not a name cannot be decrypted. The same way, a signature with a private key which is not a name cannot be checked and its message cannot be extracted.

Binders are name restrictions, message input, guarded replication, and term destructors. The set of variables that are under the scope of a binder in the thread $P$ is denoted by $bn(P)$, the set of variables that are not in the scope of a binder are denoted by $fn(P)$. We suppose that the system is well labeled, which means that no label occurs twice at the beginning of the system computation.

### 5.3.2 Semantics

As usual, the operational semantics is given by both a transition relation in Fig. 5.14 and a congruence relation in Fig. 5.13. The transition relation describes thread computation whereas the congruence relation explicitly allows for thread interactions. Communications are enabled when we have two concurrent threads such that the first one performs a reception on a channel and such that the second one performs an emission over the same channel. The result consists in launching the two corresponding continuations, just after having applied term passing. We only allow communications over channels. In case of resource fetching, the resource is still available after the communication. A test blocks thread computation if it is not applied with two equal names or with two public part of atomic keys. A term destruction tests that the arguments match, and associates the variables to the result of the destruction. The structural congruence allows for extending the scope of names, and thread meeting.

### 5.3.3 Non standard semantics

The major difficulty is to deal with terms. For that purpose, we intuitively model a memory in which each cell denotes a sub-term of the system and points to the

make this assumption to avoid testing full term equality.

| | | |
|---|---|---|
| $n$ | $\in \mathcal{N}$ | (names) |
| $x$ | $\in \mathcal{V}$ | (variables) |
| $\alpha$ | $\in \mathcal{L}_{\mathrm{n}}$ | (name labels) |
| $t$ | $\in \mathcal{L}_t$ | (term labels) |
| $p$ | $\in \mathcal{L}_a$ | (program point labels) |

| | | | |
|---|---|---|---|
| $M$ | $::=$ | $n$ | (names) |
| | $\mid$ | $x$ | (variables) |
| | $\mid$ | $\mathrm{pk}^t M$ | (public key) |
| | $\mid$ | $\mathrm{tuple}^t_n(M_1,\ldots,M_n)$ | (tuples) |
| | $\mid$ | $\mathrm{sencrypt}^t(M,M)$ | (shared-key encryption) |
| | $\mid$ | $\mathrm{pencrypt}^t(M,M)$ | (public-key encryption) |
| | $\mid$ | $\mathrm{sign}^t(M,M)$ | (signature) |
| $P$ | $::=$ | $\overline{M}^p\langle M\rangle.P$ | (output) |
| | $\mid$ | $M^p\langle x\rangle.P$ | (input) |
| | $\mid$ | $!M^p\langle x\rangle.P$ | (guarded replication) |
| | $\mid$ | $\mathbf{0}$ | (nil) |
| | $\mid$ | $P \mid P$ | (parallel composition) |
| | $\mid$ | $(\nu^\alpha n)P$ | (name restriction) |
| | $\mid$ | $\mathrm{begin}^p(M)$ | (begin event) |
| | $\mid$ | $\mathrm{end}^p(M)$ | (end event) |
| | $\mid$ | $[M = M]^p.P$ | (name matching) |
| | $\mid$ | $\mathrm{let}^p x = \mathrm{th}^n_i(M)$ in $P$ | (i-th component) |
| | $\mid$ | $\mathrm{let}^p x = \mathrm{sdecrypt}(M,M)$ in $P$ | (shared-key decryption) |
| | $\mid$ | $\mathrm{let}^p x = \mathrm{pdecrypt}(M,M)$ in $P$ | (public-key decryption) |
| | $\mid$ | $\mathrm{let}^p x = \mathrm{checksign}(M,M)$ in $P$ | (signature checking) |
| | $\mid$ | $\mathrm{let}^p x = \mathrm{getmessage}(M)$ in $P$ | (getting message) |

Figure 5.12: Syntax of the *spi*-calculus.

$$\begin{array}{rcll}
(\nu^{\alpha}x)P & \equiv & (\nu^{\alpha}y)P[x \leftarrow y] \quad \text{if } y \notin fn(P) & (\alpha\text{-conversion}) \\
P \mid Q & \equiv & Q \mid P & (\text{commutativity}) \\
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & (\text{associativity}) \\
P \mid \mathbf{0} & \equiv & P & (\text{end of a thread}) \\
(\nu^{\alpha}x)\mathbf{0} & \equiv & \mathbf{0} & (\text{garbage collecting}) \\
(\nu^{\alpha}x)(\nu^{\beta}y)P & \equiv & (\nu^{\beta}y)(\nu^{\alpha}x)P \quad \text{if } x \neq y & (\text{swapping}) \\
((\nu^{\alpha}x)P) \mid Q & \equiv & (\nu^{\alpha}x)(P \mid Q) \quad \text{if } x \notin fn(Q) & (\text{extrusion})
\end{array}$$

where $x, y \in \mathscr{N}$, $\alpha, \beta \in \mathscr{L}_{\text{n}}$.

Figure 5.13: Structural congruence of the *spi*-calculus.

$$\begin{array}{rcl}
n^i\langle x\rangle.P \mid \overline{n}^j\langle m\rangle.Q & \xrightarrow{(i,j)} & P[x \leftarrow m] \mid Q \\
!n^i\langle x\rangle.P \mid \overline{n}^j\langle m\rangle.Q & \xrightarrow{(i,j)} & !n^i\langle m\rangle.P \mid P[x \leftarrow m] \mid Q \\
[n = n]^p.P & \xrightarrow{p} & P \\
[\text{pk}(n) = \text{pk}(n)]^p.P & \xrightarrow{p} & P \\
\text{let}^p x = \text{th}^n_i(\text{tuple}^t_n(M_1, \ldots, M_n)) \text{ in } P & \xrightarrow{p} & P[x \leftarrow M_i] \\
\text{let}^p x = \text{sdecrypt}(\text{sencrypt}^t(M, n), n) \text{ in } P & \xrightarrow{p} & P[x \leftarrow M] \\
\text{let}^p x = \text{pdecrypt}(\text{pencrypt}^t(M, \text{pk}(n)), n) \text{ in } P & \xrightarrow{p} & P[x \leftarrow M] \\
\text{let}^p x = \text{checksign}(\text{sign}^t(M, n), \text{pk}(n)) \text{ in } P & \xrightarrow{p} & P[x \leftarrow M] \\
\text{let}^p x = \text{getmessage}(\text{sign}^t(M, n)) \text{ in } P & \xrightarrow{p} & P[x \leftarrow M]
\end{array}$$

$$\frac{P \xrightarrow{\lambda} Q}{(\nu^{\alpha}x)P \xrightarrow{\lambda} (\nu^{\alpha}x)Q} \qquad \frac{P' \equiv P \quad P \xrightarrow{\lambda} Q \quad Q \equiv Q'}{P' \xrightarrow{\lambda} Q'} \qquad \frac{P \xrightarrow{\lambda} P'}{P \mid Q \xrightarrow{\lambda} P' \mid Q}$$

where $m, n \in \mathscr{N}$, $x \in \mathscr{V}$, $i, j, p \in \mathscr{L}_a$, $t \in \mathscr{L}_t$, and $\lambda \in \mathscr{L}_a^*$.

Figure 5.14: reduction relation of the *spi*-calculus.

children of this sub-term. This way, we introduce term threads: each sub-term in the system is associated with a thread. The interface of this thread contains one variable for each argument of the root term constructors. These variables are associated with the child sub-term addresses. Moreover, a term thread may compute one partial interaction which has the same name as the corresponding term constructor. Term threads are not consumed during computation and launch no continuation, so the type of their partial interaction names is *migration*. Names are terminal. So we introduce for each name a term thread with an empty interface. This thread may compute a *migration* partial interaction named *name*. Such threads allow for run-time type checking.

#### 5.3.3.1 Partial interaction names

We define the set of the partial interaction names as follows:

$$\begin{array}{c}\{input;output;fetch;match\text{-}names;match\text{-}pk;name;pk\}\\ \cup\{sencrypt;pencrypt;sign;sdecrypt;pdecrypt;checksign;getmessage\}\\ \cup\{tuple_n \mid n\in\mathbb{N}\}\\ \cup\{th_{i,n}\mid 1\le i\le n\}.\end{array}$$

We define partial interaction type and arities as follows:

$$Ari=\begin{cases} input & \mapsto (1,1)\\ output & \mapsto (2,0)\\ fetch & \mapsto (1,1)\\ match\text{-}names & \mapsto (2,0)\\ match\text{-}pk & \mapsto (2,0)\\ name & \mapsto (0,0)\\ pk & \mapsto (1,0)\\ tuple_n & \mapsto (n,0)\\ sencrypt & \mapsto (2,0)\\ pencrypt & \mapsto (2,0)\\ sign & \mapsto (2,0)\\ th_{i,n} & \mapsto (1,1)\\ sdecrypt & \mapsto (2,1)\\ pdecrypt & \mapsto (2,1)\\ checksign & \mapsto (2,1)\\ getmessage & \mapsto (1,1),\end{cases} \qquad type=\begin{cases} input & \mapsto computation\\ output & \mapsto computation\\ fetch & \mapsto replication\\ match\text{-}names & \mapsto computation\\ match\text{-}pk & \mapsto computation\\ name & \mapsto migration\\ pk & \mapsto migration\\ tuple_n & \mapsto migration\\ sencrypt & \mapsto migration\\ pencrypt & \mapsto migration\\ sign & \mapsto migration\\ th_{i,n} & \mapsto computation\\ sdecrypt & \mapsto computation\\ pdecrypt & \mapsto computation\\ checksign & \mapsto computation\\ getmessage & \mapsto computation.\end{cases}$$

A partial interaction named *input* denotes a thread that is waiting for a term on a channel. Such a partial interaction is a regular computation, it has one parameter which is the name of the channel on which the term is waited for. When receiving the term, it declares one variable and associates it with the received term. A partial interaction named *output* denotes a thread that is sending a term on channel. Such a partial interaction is a regular computation, it has two parameters which are the name on which the term is sent and the communicated term; it bounds no variable. A partial interaction named *fetch* denotes a thread that may replicate itself when receiving a term on a channel. This is a replication. It requires one parameter which is the name on which the reception is made and bounds one new variable to the received term. A partial interaction named *match-names* denotes a thread that is testing whether two names are equal, or not. A partial interaction named *match-public* denotes a thread that is testing whether two terms denote the public part of the same atomic key. It is a regular computation that requires two arguments, one for each tested name. It bounds no variable. A partial interaction named *name* denotes a term which is a name. This term allows for the computation of other threads, without being modified: it is a migration. Moreover, it has no argument and bounds no variable. A partial interaction named *pk* denotes a term that is the public part of a name. It is not computed, but allows for the computation of other threads. Thus, it is a migration. Moreover, it requires one argument which is the name and bounds no variable. A partial interaction named $tuple_n$ denotes a term which is a tuple of $n$ terms. It is a migration. Moreover it requires $n$ arguments each of which denotes a term component and it bounds no variable. A partial interaction named *sencrypt*, *pencrypt*, or *sign* denotes a term which is an encryption (that respectively uses *shared-key* or *public-key*) or a signature. This term allows for the computation of the other threads: it is a migration. It requires two arguments: the first one is a term denoting the message and the second one is a term denoting the key. It bounds no variable. A partial interaction named $th_{i,n}$ denotes a thread that extracts the $i$-th component of an $n$-tuple. It is a regular computation that requires one argument, the $n$-tuple and associates a new variable with the $i$-th component of this tuple. A partial interaction named *sdecrypt*, *pencrypt*, or *checksign* denotes a thread which tries to inverse an encryption or a signature. It requires two arguments the first of which is a term that denotes the cypher-text and the second of which is a term that denotes the key. In case of success, it associates a new variable with the plain message. A partial interaction named *getmessage* denotes a thread that extracts the plain message of a signed message. It requires one argument which is the term that denotes the signed message. In case of success, it associates a new variable with the original message.

#### 5.3.3.2 Formal rules

We now define the formal rules that implement reduction steps. The rule *com* which is described in Fig. 5.15 implements the communication. It requires three threads: the first two threads denote the communicating threads: the first one waits for a message (*input*) and the second one sends a message (*output*); the third thread is used to check that the communication is done over a channel name (*name*). The synchronization condition $X_1^1 = X_1^2$ ensures that the first parameters of each component are associated with the same value. The synchronization condition $X_1^1 = I^3$ ensures that this value is the name of a channel. At last, when interacting, the first bound variable of the receiver is associated with the value linked with the second parameter of the message sender: this models term passing. There is no broadcast communication.

$$com = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto input \\ 2 \mapsto output \\ 3 \mapsto name; \end{cases}$
2. $synchronization = \{X_1^1 = X_1^2; X_1^1 = I^3\};$
3. $communication = [Y_1^1 \leftarrow X_2^2, \forall i \in [\![1;n]\!]\ ];$
4. $global = \emptyset.$

Figure 5.15: Formal rule for communication in the *spi*-calculus.

The rule *fetch* which is described in Fig. 5.16 implements the replication of a guarded resource. It is exactly the same as the communication rule, except the fact that the second thread computes a replication $(fetch)$. The facts that the resource is still available after the computation step and that a new marker is associated with the new instance are dealt with systematically by the generic operational semantics (Cf. Sect. 4.4.2). That is why we did not need to describe them explicitly in the rule.

The rule *name-matching* which is described in Fig. 5.17 implements name matching. It requires two threads: the first thread denotes the thread that perform the matching (*match-names*); the second one denotes the name (*name*). The formal variable $X_1^1$ and $X_2^1$ denote pointers to the first and the second arguments of the matching guard. The formal variable $I^2$ denotes the name. The synchronization conditions $X_1^1 = I^2$ and $X_2^1 = I^2$ ensure that both parameters of the matching guard

$$fetch = (3, component, synchronization, communication, global)$$

where:

1. $component = \begin{cases} 1 \mapsto fetch \\ 2 \mapsto output \\ 3 \mapsto name; \end{cases}$

2. $synchronization = \{X_1^1 = X_1^2; X_1^1 = I^3\};$

3. $communication = [Y_1^1 \leftarrow X_2^2, \forall i \in [\![1;n]\!]\ ];$

4. $global = \emptyset.$

Figure 5.16: Formal rule for guarded replication in the *spi*-calculus.

are associated with the name. There is no local communication and no broadcast communication.

$$name\text{-}matching = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto match\text{-}names \\ 2 \mapsto name \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_2^1 = I^2\};$

3. $communication = \emptyset;$

4. $global = \emptyset.$

Figure 5.17: Formal rule for name matching in the *spi*-calculus.

The rule *public-key-matching* which is described in Fig. 5.18 implements public key matching. It requires four threads: the first thread denotes the thread that perform the matching (*match-pk*); the second one denotes the private part of the key: it must be a name (*name*); the third and the fourth one denote two instances of the public part of the key $(pk)$. The formal variable $X_1^1$ and $X_2^1$ denote pointers to the first and the second arguments of the matching guard. The formal variable $I^2$ denotes the name. The formal variables $I^3$ and $I^4$ denote the instances of the public part of the name. The formal variable $X_1^3$ denotes a pointer to the private part of the first instance of the public key. The formal variable $X_1^4$ denotes a pointer to the private part of the second instance of the public key. The test may be passed if and only if:

- the first argument of the guard points to the first instance of the public key (i.e. $X_1^1 = I^3$);
- the second argument of the guard points to the second instance of the public key (i.e. $X_2^1 = I^4$);
- the argument of the first instance of the public key points to the private key (i.e. $X_1^3 = I_2$);
- the argument of the second instance of the public key points to the private key (i.e. $X_1^4 = I_2$).

There is no communication and no broadcast communication.

$$\mathit{public\text{-}name\text{-}matching} = (2, \mathit{component}, \mathit{synchronization}, \mathit{communication}, \mathit{global})$$

where

1. $\mathit{component} = \begin{cases} 1 \mapsto \mathit{match\text{-}pk} \\ 2 \mapsto \mathit{name} \\ 3 \mapsto \mathit{pk} \\ 4 \mapsto \mathit{pk} \end{cases}$
2. $\mathit{synchronization} = \{X_1^1 = I^3; X_2^1 = I^4; X_1^3 = I_2; X_1^4 = I_2\}$;
3. $\mathit{communication} = \emptyset$;
4. $\mathit{global} = \emptyset$.

Figure 5.18: Formal rule for public key matching in the *spi*-calculus.

*Remark* 5.3.1. We could have proposed other rules to extend matching to any syntactic skeleton.

The formal rule $\mathit{proj}_i^n$ which is described in Fig. 5.19 denotes the projection of an $n$-tuple according to its $i$-th component. It requires two threads: the first one denotes the thread which performs the projection ($\mathit{th}_{i,n}$) and the second one denotes the term which is projected ($\mathit{tuple}_n$). The formal variable $X_1^1$ denotes a pointer to the projected term. The formal variable $X_j^2$ denotes a pointer to the $j$-th component of the term and the variable $I^2$ denotes the term identity. The reduction step is enabled if and only if the pointer of the thread point to the tuple (i.e. $X_1^1 = I^2$). In such a case, the thread declares a fresh variable which is denoted by the formal variable $Y_1^1$ and which is associated with the $i$-th component of the tuple (i.e. $[Y_1^1 \leftarrow X_i^2]$). There is no broadcast communication.

$$proj_i^n = (2, component, synchronization, communication, global)$$

where:

1. $component(1) = th_{i,n}$ and $component(2) = tuple_n$;
2. $synchronization = \{X_1^1 = I^2\}$;
3. $communication = [Y_1^1 \leftarrow X_i^2]$;
4. $global = \emptyset$.

Figure 5.19: Formal rule for tuple destruction.

The formal rule *sdecrypt* which is described in Fig. 5.20 denotes a shared-key decryption. It requires three threads: the first one denotes the thread which performs the decryption (*sdecrypt*), the second one denotes the term that is decrypted (*sencrypt*), the third one denotes the key that is used for both the encryption and the decryption, this key must be a name (*name*). The formal variables $X_1^1$ and $X_2^1$ denote pointers to the first and the second arguments of the decryption. The formal variables $X_1^2$ and $X_2^2$ denote pointers to the first and the second arguments of the encryption. The formal variable $I^2$ denotes the encryption term identity, and the formal variable $I^3$ denotes the identity of the key. The decryption is enabled if and only if:

- the first argument of the decryption points to the encrypted term (i.e. $X_1^1 = I^2$);
- the second argument of the decryption points to the key (i.e. $X_2^1 = I^3$);
- the second argument of the encryption points to the key (i.e. $X_2^2 = I^3$).

In such a case, the thread declares a fresh variable, which is denoted by the formal variable $Y_1^1$ and which is associated with the plain message (i.e. $[Y_1^1 \leftarrow X_1^2]$). There is no broadcast communication.

The formal rule *pdecrypt* which is described in Fig. 5.21 denotes a public-key decryption. It requires four threads: the first one denotes the thread which performs the decryption (*pdecrypt*), the second one denotes the term that is decrypted (*pencrypt*), the third one denotes the public part of the key that has been

$$sdecrypt = (3, component, synchronization, communication, global)$$

where:

1. $component = \begin{cases} 1 \mapsto sdecrypt \\ 2 \mapsto sencrypt \\ 3 \mapsto name; \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_2^1 = I^3; X_2^2 = I^3\}$;

3. $communication = [Y_1^1 \leftarrow X_1^2]$;

4. $global = \emptyset$.

Figure 5.20: Formal rule for shared-key decryption.

used for the encryption (*pk*) and the fourth one denotes the private part of the key: this must be a name (*name*). The formal variables $X_1^1$ and $X_2^1$ denote pointers to the first and the second arguments of the decryption. The formal variables $X_1^2$ and $X_2^2$ denote pointers to the first and the second arguments of the encryption. The formal variable $X_1^3$ denotes a pointer to the private part of the key. The formal variable $I^2$ denotes the encryption term identity, the formal variable $I^3$ denotes the identity of the key public part, and the formal variable $I^4$ denotes the identity of the private part of the key. The decryption is enabled if and only if:

- the first argument of the decryption points to the encrypted term (i.e. $X_1^1 = I^2$);
- the second argument of the decryption points to the private part of the key (i.e. $X_2^1 = I^4$);
- the second argument of the encryption points to the public part of the key (i.e. $X_2^2 = I^3$);
- the argument of the key public part points to the private part of the key (i.e. $X_1^3 = I^4$).

In such a case, the thread declares a fresh variable, which is denoted by the formal variable $Y_1^1$ and which is associated with the plain message (i.e. $[Y_1^1 \leftarrow X_1^2]$). There is no broadcast communication.

The formal rule *checksign* which is described in Fig. 5.22 denotes a signature checking. It requires four threads: the first one denotes the thread which checks the signature (*checksign*), the second one denotes the term that is signed (*sign*), the third one denotes the public part of the key that has been used for the signature

$$pdecrypt = (4, component, synchronization, communication, global)$$

where:

1. $component = \begin{cases} 1 \mapsto pdecrypt \\ 2 \mapsto pencrypt \\ 3 \mapsto pk \\ 4 \mapsto name; \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_2^1 = I^4; X_2^2 = I^3; X_1^3 = I^4\};$

3. $communication = [Y_1^1 \leftarrow X_1^2];$

4. $global = \emptyset.$

Figure 5.21: Formal rule for public-key decryption.

($pk$) and the fourth one denotes the private part of the key: this must be a name ($name$). The formal variables $X_1^1$ and $X_2^1$ denote pointers to the first and the second arguments of the signature checking. The formal variables $X_1^2$ and $X_2^2$ denote pointers to the first and the second arguments of the signed message. The formal variable $X_1^3$ denotes a pointer to the private part of the key. The formal variable $I^2$ denotes the signed message term identity, the formal variable $I^3$ denotes the identity of the key public part, and the formal variable $I^4$ denotes the identity of the private part of the key. The signature checking is enabled if and only if:

- the first argument of the signature checking points to the signed message (i.e. $X_1^1 = I^2$);
- the second argument of the signature checking points to the public part of the key (i.e. $X_2^1 = I^3$);
- the second argument of the signed message points to the private part of the key (i.e. $X_2^2 = I^4$);
- the argument of the key public part points to the private part of the key (i.e. $X_1^3 = I^4$);

In such a case, the thread declares a fresh variable, which is denoted by the formal variable $Y_1^1$ and which is associated with the plain message (i.e. $[Y_1^1 \leftarrow X_1^2]$). There is no broadcast communication.

The formal rule *getmessage* which is described in Fig. 5.23 denotes the signature removing. It requires three threads: the first one denotes the thread which removes the signature (*getmessage*), the second one denotes the term that is signed

$$checksign = (4, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto checksign \\ 2 \mapsto sign \\ 3 \mapsto pk \\ 4 \mapsto name; \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_2^1 = I^3; X_2^2 = I^4; X_1^3 = I^4\};$

3. $communication = [Y_1^1 \leftarrow X_1^2];$

4. $global = \emptyset.$

Figure 5.22: Formal rule for signature verification.

(*sign*), the third one denotes the private part of the key that signs the message, it must be a name (*name*). The formal variable $X_1^1$ denotes a pointer to the argument of the signature removing. The formal variables $X_1^2$ and $X_2^2$ denote pointers to the first and the second arguments of the signed message. The formal variable $I^2$ denotes the signed message term identity, the formal variable $I^3$ denotes the identity of the private part of the key. The signature checking is enabled if and only if:

- the argument of the signature removing points to the signed message (i.e. $X_1^1 = I^2$);

- the second argument of the signed message points to the private part of the key (i.e. $X_2^2 = I^3$).

In such a case, the thread declares a fresh variable, which is denoted by the formal variable $Y_1^1$ and which is associated with the plain message (i.e. $[Y_1^1 \leftarrow X_1^2]$). There is no broadcast communication.

#### 5.3.3.3 Abstract syntax extraction

We now define the syntax extraction function that takes a program that describes the initial state of a mobile system in the standard syntax and extracts the abstract syntax of it. Program points are term constructors, name restrictions, term destructors and communication actions:

- the label $\alpha$ of a name restriction $(\nu^{\alpha} x).P$ is associated with the empty interface and with the following set of partial interactions:

$$\{(names, [], [], \emptyset, \{\{(\alpha, \emptyset)\}\})\};$$

$$getmessage = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto getmessage \\ 2 \mapsto sign \\ 3 \mapsto name; \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_2^2 = I^3\}$;

3. $communication = [Y_1^1 \leftarrow X_1^2]$;

4. $global = \emptyset$.

Figure 5.23: Formal rule for getting message.

- the label $p$ of a program point $Q = M^p\langle x\rangle.P$ is associated with the interface $\{X\} \cup fn(Q)$ ($X$ denotes a pointer to the term that describes the channel on which the reception is performed) and with the following set of partial interactions:
$$\{(input, [X], [x], \emptyset, \{\beta(P, E_s)\})\};$$

- the label $p$ of a program point $Q = !M^p\langle x\rangle.P$ is associated with the interface $\{X\} \cup fn(Q)$ and with the following set of partial interactions:
$$\{(fetch, [X], [x], \emptyset, \{\beta(P, E_s)\})\};$$

- the label $p$ of a program point $Q = \overline{M_1}^p\langle M_2\rangle.P$ is associated with the interface $\{X_1; X_2\} \cup fn(Q)$ and with the following set of partial interactions:
$$\{(output, [X_1; X_2], [], \emptyset, \{\beta(P, E_s)\})\};$$

- the label $p$ of a program point begin$^p(M)$ or end$^p(M)$ is associated with the interface $\{X\}$ and with the empty set of partial interactions;

- the label $p$ of a program point $Q = [M_1 = M_2]^p.P$ is associated with the interface $\{X_1; X_2\} \cup fn(Q)$ and with the following set of partial interactions:
$$\left\{ \begin{array}{l} (match\text{-}names, [X_1; X_2], [], \emptyset, \{\beta(P, E_s)\}) \\ (match\text{-}pk, [X_1; X_2], [], \emptyset, \{\beta(P, E_s)\}) \end{array} \right\};$$

- the label $t$ of a term construction $pk^t(M)$ is associated with the interface $\{X\}$ and with the following set of partial interactions:
$$\{(pk, [X], [], \emptyset, \{\{(t, \emptyset)\}\})\};$$

- the label $t$ of a term construction $\mathit{tuple}_n^t(M_1,\ldots,M_n)$ is associated with the interface $\{X_1;\ldots;X_n\}$ and with the following set of partial interactions:

$$\{(\mathit{tuple}_n,[X_1;\ldots;X_n],[],\emptyset,\{\{(t,\emptyset)\}\})\};$$

- the label $t$ of a term construction $\mathit{sencrypt}^t(M_1,M_2)$ is associated with the interface $\{X_1;X_2\}$ and with the following set of partial interactions:

$$\{(\mathit{sencrypt},[X_1;X_2],[],\emptyset,\{\{(t,\emptyset)\}\})\};$$

- the label $t$ of a term construction $\mathit{pencrypt}^t(M_1,M_2)$ is associated with the interface $\{X_1;X_2\}$ and with the following set of partial interactions:

$$\{(\mathit{pencrypt},[X_1;X_2],[],\emptyset,\{\{(t,\emptyset)\}\})\};$$

- the label $t$ of a term construction $\mathit{sign}^t(M_1,M_2)$ is associated with the interface $\{X_1;X_2\}$ and with the following set of partial interactions:

$$\{(\mathit{sign},[X_1;X_2],[],\emptyset,\{\{(t,\emptyset)\}\})\};$$

- the label $p$ of a term destruction $\mathrm{let}^p x = \mathrm{destruct}\,\mathrm{in}\,P$ is associated with the interface $\mathit{var}(\mathrm{desctruct}) \cup (\mathit{fn}(Q))$ and with the following set of partial interactions:

$$\{(\mathit{act}(\mathrm{desctruct}),\mathit{arg}(\mathrm{destruct}),[x],\emptyset,\{\beta(P,E_s)\})\},$$

where the auxiliary function *act*, *arg*, and *var* are defined as follows:

$$\mathit{act} = \begin{cases} \mathrm{th}_i^n(M) & \mapsto \mathit{th}_{i,n} \\ \mathrm{sdecrypt}(M,M) & \mapsto \mathit{sdecrypt} \\ \mathrm{pdecrypt}(M,M) & \mapsto \mathit{pdecrypt} \\ \mathrm{checksign}(M,M) & \mapsto \mathit{checksign} \\ \mathrm{getmessage}(M) & \mapsto \mathit{getmessage}; \end{cases}$$

$$\mathit{arg} = \begin{cases} \mathrm{th}_i^n(M) & \mapsto [X] \\ \mathrm{sdecrypt}(M,M) & \mapsto [X_1;X_2] \\ \mathrm{pdecrypt}(M,M) & \mapsto [X_1;X_2] \\ \mathrm{checksign}(M,M) & \mapsto [X_1;X_2] \\ \mathrm{getmessage}(M) & \mapsto [X]; \end{cases}$$

$$\mathit{var} = \begin{cases} \mathrm{th}_i^n(M) & \mapsto \{X\} \\ \mathrm{sdecrypt}(M,M) & \mapsto \{X_1;X_2\} \\ \mathrm{pdecrypt}(M,M) & \mapsto \{X_1;X_2\} \\ \mathrm{checksign}(M,M) & \mapsto \{X_1;X_2\} \\ \mathrm{getmessage}(M) & \mapsto \{X\}. \end{cases}$$

Continuation computations use mutually recursive auxiliary primitives $\beta$, $A$, and $B$. The primitive $\beta$ computes the continuation[2]. It is defined inductively over the standard syntax of the continuation in Fig. 5.24. Primitives $A$ and $B$ are used to store terms. The primitive $A$ allocates a memory location for each sub-term. The primitive $B$ returns the address of the term root. The primitive $\beta$ is defined inductively in Fig. 5.24. Primitives $A$ and $B$ are defined in Fig. 5.25.

$$\begin{array}{rcl}
\beta(\overline{M_1}^p\langle M_2\rangle.P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X_i\mapsto B(M_i,E_s)])\}\\ A(M_1,E_s);A(M_2,E_s)\end{array}\right\}\\
\beta(M^p\langle x\rangle.P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta(!M^p\langle x\rangle.P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta(\mathbf{0},E_s) & = & \emptyset\\
\beta(P\mid Q,E_s) & = & \beta(P,E_s)\cup\beta(Q,E_s)\\
\beta((\nu^\alpha n)P) & = & \beta(P,E_s[n\mapsto\alpha])\cup\{(\alpha,\emptyset)\}\\
\beta(\mathrm{begin}^p(M),E_s) & = & \left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta(\mathrm{end}^p(M),E_s) & = & \left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta([M_1=M_2]^p.P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X_i\mapsto B(M_i,E_s)])\};\\ A(M_1,E_s);A(M_2,E_s)\end{array}\right\}\\
\beta(\mathrm{let}^p x=\mathrm{th}_i^n(M)\text{ in }P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta(\mathrm{let}^p x=\mathrm{getmessage}(M)\text{ in }P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X\mapsto B(M,E_s)])\};\\ A(M,E_s)\end{array}\right\}\\
\beta(\mathrm{let}^p x=\mathrm{sdecrypt}(M_1,M_2)\text{ in }P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X_i\mapsto B(M_i,E_s)])\};\\ A(M_1,E_s);A(M_2,E_s)\end{array}\right\}\\
\beta(\mathrm{let}^p x=\mathrm{pdecrypt}(M_1,M_2)\text{ in }P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X_i\mapsto B(M_i,E_s)])\};\\ A(M_1,E_s);A(M_2,E_s)\end{array}\right\}\\
\beta(\mathrm{let}^p x=\mathrm{checksign}(M_1,M_2)\text{ in }P,E_s) & = & \bigcup\left\{\begin{array}{l}\{(p,E_s[X_i\mapsto B(M_i,E_s)])\};\\ A(M_1,E_s);A(M_2,E_s)\end{array}\right\}
\end{array}$$

Figure 5.24: Abstract syntax extraction.

The set of initial states is defined as follows:

$$\beta(\mathscr{S},\emptyset).$$

[2]We avoid defi ning a set of continuations, since we use no non-deterministic choice

$$\begin{array}{rcl}
A(n,E_s) & = & \emptyset \\
A(x,E_s) & = & \emptyset \\
A(\text{tuple}^t_n(M_1,\ldots,M_n),E_s) & = & \{(t,[X_i \mapsto B(M_i,E_s))]\} \cup \bigcup\{A(M_i,E_s) \mid 1 \le i \le n\} \\
A(\text{pk}^t(M),E_s) & = & \{(t,[X_1 \mapsto B(M,E_s)])\} \cup A(M,E_s) \\
A(\text{sencrypt}^t(M_1,M_2),E_s) & = & \{(t,[X_i \mapsto B(M_i,E_s)])\} \cup A(M_1,E_s) \cup A(M_2,E_s) \\
A(\text{pencrypt}^t(M_1,M_2),E_s) & = & \{(t,[X_i \mapsto B(M_i,E_s))]\} \cup A(M_1,E_s) \cup A(M_2,E_s) \\
A(\text{sign}^t(M_1,M_2),E_s) & = & \{(t,[X_i \mapsto B(M_i,E_s))]\} \cup A(M_1,E_s) \cup A(M_2,E_s)
\end{array}$$

$$\begin{array}{rcl}
B(n,E_s) & = & E_s(n) \\
B(x,E_s) & = & E_s(x) \\
B(\text{tuple}^t_n(M_1,\ldots,M_n),E_s) & = & t \\
B(\text{pk}^t(M),E_s) & = & t \\
B(\text{sencrypt}^t(M_1,M_2),E_s) & = & t \\
B(\text{pencrypt}^t(M_1,M_2),E_s) & = & t \\
B(\text{sign}^t(M_1,M_2),E_s) & = & t
\end{array}$$

Figure 5.25: Term allocation.

### 5.3.4 Correspondence

**Proposition 5.3.2.** *For each system, the extracted syntax satisfies the sufficient properties in Def. 4.5.11. Thus, allocated markers are unambiguous.*

The correspondence between the standard and the non standard semantics is established as follows:

**Theorem 5.3.3.** *The standard and the non standard semantics are in strong bisimulation.*

## 5.4 Revisiting the *ambient* calculus

In this section, we consider the same standard semantics as in Chap. 3 and we show how we may specify the same non standard semantics in our meta language.

The encoding of the ambients requires a careful treatment of locations. The location of ambients and of threads will be stored in their environment, by using a special variable *loc*. Migrations are modeled by updating the value that is associated with the variable *loc*. The last difficulty is to model ambient dissolution where the whole content of an ambient is moved inside another ambient. This is modeled by using a broadcast communication in order to replace any instance of the dissolved ambient identity with the dissolved ambient location.

### 5.4.1 Non standard semantics

#### 5.4.1.1 Partial interaction names

We define the set of partial interaction names as follows:

$$\{input; fetch; output; in; out; open; ropen; mov\text{-}ambient; dis\text{-}ambient\};$$

Their types and their arities are given as follows:

$$Ari = \begin{cases} input & \mapsto (1,1) \\ fetch & \mapsto (1,1) \\ output & \mapsto (2,0) \\ in & \mapsto (2,0) \\ out & \mapsto (2,0) \\ open & \mapsto (2,0) \\ ropen & \mapsto (2,0) \\ mov\text{-}ambient & \mapsto (2,1) \\ dis\text{-}ambient & \mapsto (2,0); \end{cases} \quad type = \begin{cases} input & \mapsto computation \\ fetch & \mapsto replication \\ output & \mapsto computation \\ in & \mapsto computation \\ out & \mapsto computation \\ open & \mapsto computation \\ ropen & \mapsto replication \\ mov\text{-}ambient & \mapsto migration \\ dis\text{-}ambient & \mapsto computation. \end{cases}$$

The partial interaction name *input* denotes a thread that is waiting for an ambient name. Such a partial interaction is a regular thread computation, it has one parameter which is the location of the thread and it bounds one variable. The partial interaction name *fetch* denotes a thread that replicates itself when receiving an ambient name. Such a partial interaction is a replication, it has one parameter which is the location of the thread and it bounds one variable. The partial interaction name *output* denotes a thread that sends an ambient name. Such a partial interaction is a regular thread computation, it has two parameters: the first one is the location of the thread and the second one is the name which is communicated. This partial interaction bounds no variable. The partial interaction name *in* denotes a thread that provides to its surrounding ambient the capability to move inside another ambient. The partial interaction name *out* denotes a thread that provides to its parent ambient the capability to exit its grand-parent ambient. The partial interaction name *open* denotes a thread that may dissolve a concurrent ambient. The partial interaction name *ropen* denotes a thread that may replicate itself when dissolving an ambient. The partial interactions named *in*, *out*, and *open* are regular computations whereas the partial interactions named *ropen* are replications. All these interactions have two parameters which are the location of the thread and the name on which the capability may be applied with. They bound no variables.

Ambients may migrate and may be dissolved. In order to compute the result of a migration or of a dissolution, the ambient must check some assumptions about its identity, its location, and its name. In the case of a migration, it redefines its location and in the case of a dissolution, it replace each occurrence of its identity in the whole system with the location of its parent. Thus, we introduce two other partial interaction names: The name *mov-ambient* denotes the capability of migrating. Such a partial interaction has two parameters: the first one is the location of the ambient before the migration and the second one is the name of the ambient. It bounds one variable which is the updated location after the migration. The name *dis-ambient* denotes the capability of being dissolved. Such a partial interaction has two parameters: the first one is the location of the ambient and the second one the ambient name. It bounds no variable.

#### 5.4.1.2 Formal rules

We now define the formal rules that implement migrations, dissolutions, and local communications. The rule *in* which is described in Fig. 5.26 implements the migration of an ambient inside its parent ambient. It requires three threads: the first

$$in = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto mov\text{-}ambient \\ 2 \mapsto mov\text{-}ambient \\ 3 \mapsto in; \end{cases}$

2. $synchronization = \{X_1^1 = X_1^2; X_1^3 = I^1; X_2^2 = X_2^3\};$

3. $communication = [Y_1^1 \leftarrow I^2, Y_1^2 \leftarrow X_1^2];$

4. $global = \emptyset.$

Figure 5.26: Formal rule for in migration.

one is the migrating ambient (*mov-ambient*), the second one is the target ambient (*mov-ambient*) and the third one denotes a thread (*in*) that provides to the first ambient the capability to migrate into the second one. The formal variables $X_1^1$, $X_1^2$, and $X_1^3$ respectively denote the locations of the migrating ambient, of the target ambient, and of the thread. The formal variable $X_2^2$ denotes the name of the target ambient and the formal variable $X_2^3$ denotes the name with which the capability applies. The name of the migrating ambient is not relevant for the computation step. The formal variables $I^1$ and $I^2$ respectively denote the identity of the migrating ambient and of the target ambient. The migration is enabled if and only if:

1. the migrating and the target ambients are located in the same ambient (i.e. $X_1^1 = X_1^2$);
2. the migrating ambient contains the thread (i.e. $X_1^3 = I^1$);
3. the capability applies with the name of the target ambient (i.e. $X_2^2 = X_2^3$).

At last, when interacting, the migrating ambient updates its location which is now the identity of the target ambient. The formal variable $Y_1^1$ denotes the new location of the migrating ambient. Thus, the variable $Y_1^1$ is associated with the variable $I^2$. The second ambient does not move. The formal variable $Y_1^2$ denotes the location of the second ambient. Thus, the variable $Y_1^2$ is associated with the variable $X_1^2$. There is no broadcast communication.

The rule *out* which is described in Fig. 5.27 implements the migration of an ambient out of its parent ambient. It requires three threads: the first one

$$out = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto \textit{mov-ambient} \\ 2 \mapsto \textit{mov-ambient} \\ 3 \mapsto \textit{out}; \end{cases}$
2. $synchronization = \{X_1^1 = I^2; X_1^3 = I^1; X_2^2 = X_2^3\}$;
3. $communication = [Y_1^1 \leftarrow X_1^2, Y_1^2 \leftarrow X_1^2]$;
4. $global = \emptyset$.

Figure 5.27: Formal rule for out migration.

is the migrating ambient (*mov-ambient*), the second one is the exited ambient (*mov-ambient*) and the third one denotes a thread (*out*) that provides to the first ambient the capability to migrate out of the second one. The formal variables $X_1^1$, $X_1^2$, and $X_1^3$ respectively denote the locations of the migrating ambient, of the exited ambient, and of the thread. The formal variable $X_2^2$ denotes the name of the exited ambient and the formal variable $X_2^3$ denotes the name with which the capability applies. The name of the migrating ambient is not relevant for the computation step. The formal variables $I^1$ and $I^2$ respectively denote the identity of the migrating ambient and of the exited ambient. The migration is enabled if and only if:

1. the exited ambient contains the migrating ambient (i.e. $X_1^1 = I_2$);

2. the migrating ambient contains the thread (i.e. $X_1^3 = I^1$);

3. the capability applies with the name of the target ambient (i.e. $X_2^2 = X_2^3$).

At last, when interacting, the migrating ambient updates its location which is now the location of the target ambient. The formal variable $Y_1^1$ denotes the new location of the migrating ambient. Thus, the variable $Y_1^1$ is associated with the variable $X_1^2$. The second ambient does not move. The formal variable $Y_1^2$ denotes the location of the second ambient. Thus, the variable $Y_1^2$ is associated with the variable $X_1^2$. There is no broadcast communication.

The rules *open* and *ropen* which are described in Fig. 5.28 and Fig. 5.29 implement the dissolution of an ambient and the dissolution of an ambient with the replication of the dissolving thread.

$$open = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto open \\ 2 \mapsto dis\text{-}ambient \end{cases}$

2. $synchronization = \{X_1^1 = X_1^2; X_2^1 = X_2^2\}$;

3. $communication = \emptyset$;

4. $global = [I^2 \mapsto X_1^2]$.

Figure 5.28: Formal rule for dissolution.

$$ropen = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto ropen \\ 2 \mapsto dis\text{-}ambient \end{cases}$

2. $synchronization = \{X_1^1 = X_1^2; X_2^1 = X_2^2\}$;

3. $communication = \emptyset$;

4. $global = [I^2 \mapsto X_1^2]$.

Figure 5.29: Formal rule for dissolution with replication.

They require two threads: the first one denotes a thread (*open* in the case when the thread is not replicated, *ropen* otherwise) which may dissolve an ambient and the second one is the ambient. The formal variables $X_1^1$ and $X_1^2$ respectively denote the locations of the thread and of the ambient. The formal variable $X_2^1$ denotes the name with which the capability applies and the formal variable $X_2^2$ denotes the name of the ambient. The formal variable $I^2$ denotes the ambient identity. The dissolution is enabled if and only if:

1. the thread and the ambient are located inside the same location (i.e. $X_1^1 = X_2^1$);
2. the capability applies with the name of the ambient (i.e. $X_2^1 = X_2^2$).

There is no name passing communication, but any instance of the ambient identity in the whole system must be updated with the old location of the ambient. The dissolved ambient is automatically removed because the name of the partial interaction that it computes is of type *computation* and because it has an empty continuation. If a replication is required, it is automatically handled thanks to the type of the partial interaction that is computed by the thread.

The rules *com* and *fetch* which are described in Fig. 5.30 and Fig. 5.31 implement the local communication between two threads and the local communication with replication. They require two threads: the first one denotes a thread that is

$$com = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto input \\ 2 \mapsto output \end{cases}$
2. $synchronization = \{X_1^1 = X_1^2\}$;
3. $communication = [Y_1^1 \leftarrow X_2^2]$;
4. $global = \emptyset$.

Figure 5.30: Formal rule for communication.

waiting for an ambient name (*input* in the case when the thread is not replicated, *fetch* otherwise) and the second one denotes a thread that is sending an ambient name. The formal variables $X_1^1$ and $X_1^2$ denote the locations of these threads. The formal variable $X_2^1$ denotes the name which is sent. The communication is enabled if and only if the thread and the ambient are located inside the same location (i.e. $X_1^1 = X_2^1$). In such a case, a new variable is declared and associated with

$$fetch = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto fetch \\ 2 \mapsto output \end{cases}$
2. $synchronization = \{X_1^1 = X_1^2\}$;
3. $communication = [Y_1^1 \leftarrow X_2^2]$;
4. $global = \emptyset$.

Figure 5.31: Formal rule for resource fetching.

the output ambient name. The new variable is denoted by the formal variable $Y_1^1$. Name passing is described by associating the formal variable $Y_1^1$ with the formal variable $X_2^2$. These is no broadcast communication.

#### 5.4.1.3 Abstract syntax extraction

We now define the syntax extraction function that takes a program describing the initial state of a mobile system in the standard syntax and extract the abstract syntax of it.

Program points are ambient creation and actions. The abstract syntax maps each program point label $l$ to the following set of partial interactions according to the syntax of the program point:

- the label $l$ of a program point $n^l[P]$ is associated with the interface $\{loc; n\}$ and the following set of partial interactions:

$$\left\{ \begin{array}{l} (mov\text{-}ambient, [loc; n], [loc], \emptyset, \{\{(l, \emptyset)\}\}); \\ (dis\text{-}ambient, [loc; n], [], \emptyset, \{\emptyset\}) \end{array} \right\}.$$

- the label $l$ of a program point $a^l n.P$ with $a \in \{in, out, open, !open\}$ is associated with the interface $\{loc; n\} \cup (fn(P))$ and with the following set of partial interactions:

$$\{(a, [loc; n], [], \emptyset, \{\beta(P, \emptyset)\})\};$$

- the label $l$ of a program point $(n)^l.P$ or $!(n)^l.P$ is associated with the interface $\{loc\} \cup fn(P) \setminus \{n\}$ and with the following set of partial interactions:

$$\{(act, [loc], [n], \emptyset, \{\beta(P, \emptyset)\})\},$$

where $act = input$ in the case when the program point matches $(n)^l.P$ and $act = fetch$ otherwise;

- the label $l$ of a program point $\langle n\rangle^l$ is associated with the interface $\{loc; n\}$ and with the following set of partial interactions:

$$\{(output, [loc; n], [], \emptyset, \{\emptyset\})\}.$$

Continuation computations use an auxiliary primitive $\beta$. This primitive computes the continuation[3]. It is defined inductively over the standard syntax of the continuation, as follows:

$$\begin{array}{rcl}
\beta(n^i[P], E_s) & = & \beta(P, E_s[loc \mapsto i]) \cup \{(i, E_s)\} \\
\beta(P \mid Q, E_s) & = & \beta(P, E_s) \cup \beta(Q, E_s) \\
\beta((\nu^l n)P, E_s) & = & \beta(P, E_s[n \mapsto l]) \\
\beta(M, E_s) & = & \{(M, E_s)\} \\
\beta(io, E_s) & = & \{(io, E_s)\} \\
\beta(\mathbf{0}, E_s) & = & \emptyset
\end{array}$$

The set of initial states is defined as follows:

$$\beta(\mathscr{S}, [loc \mapsto (\mathrm{top}, \varepsilon)]),$$

where the location $(\mathrm{top}, \varepsilon)$ denotes the address of the top level ambient.

### 5.4.2 Correspondence

**Proposition 5.4.1.** *For each system, the extracted syntax satisfies the sufficient properties in Def. 4.5.11. Thus, allocated markers are unambiguous.*

The correspondence between the standard and the non standard semantics is established as follows:

**Theorem 5.4.2.** *The standard and the non standard semantics are in strong bisimulation.*

## 5.5 Encoding BIO-*ambients*

The BIO-*ambients* [68] allow the description of biological systems: A BIO-*ambient* can be seen as a compartment that may describe a cell, an organelle, or a vesicle. BIO-*ambients* are bounded places which are delimited by membranes.

[3]We avoid defi ning a set of continuations, since we use no non-deterministic choice.

These compartments are anonymous and they may contain other compartments and threads. These threads provide their surrounding ambient some capability to transform the topology of the biological system. As in the *ambient* calculus [17], ambients may enter into a sibling ambient or exit their surrounding ambient. There is no dissolution, but sibling ambients may merge their content and become an unique compartment. As in the safe-ambients calculus [52], any modification of the ambients hierarchy requires two threads: the first one in the migrating ambient and the second one in the target ambient for migration or one thread in each merging ambient. These threads must perform the correct capability over the same name to enable the corresponding computation step. As in the seal-calculus [75] or in the boxed ambients [13, 12], threads may pass some names through ambient boundaries. These communications are channeled, so that two threads may only communicate if they communicate via the same channel name and if their location satisfies the good assumption.

We use a version of the BIO-*ambients* with explicit recursion *à la join*-calculus [40]. Whenever a recursive process is defined, recursive instances are always launched at the locations of the threads which unfold the recursion.

### 5.5.1 Syntax

Let $\mathcal{N}_c$ and $\mathcal{N}_p$ be two disjoint sets of names. The set $\mathcal{N}_c$ is a set of channel names, whereas the set $\mathcal{N}_p$ is a set of recursive variable names. Let $\mathcal{L}_a$ be a set of thread labels, $\mathcal{L}_c$ be a set of channel name labels, $\mathcal{L}_{amb}$ be a set of ambient labels, and $\mathcal{L}_p$ be a set of recursive variable labels. We suppose that the sets $\mathcal{L}_a$, $\mathcal{L}_c$, $\mathcal{L}_{amb}$, and $\mathcal{L}_p$ are pair-wise distinct. Moreover, we set $\mathcal{L}_{\mathrm{p}} = \mathcal{L}_a \cup \mathcal{L}_{amb}$ and $\mathcal{L} = \mathcal{L}_{\mathrm{p}} \cup \mathcal{L}_c \cup \mathcal{L}_p$. The syntax of BIO-*ambients* is described in Fig. 5.32. Program points are communication choices, capability choices, recursive definitions, and recursion unfolders. We assume that any label occurs at most once in the syntax of the initial system. A definition $(\text{let}^p\ A^{\alpha}\langle n_1,\dots,n_q\rangle = Q \text{ in } P)$ defines a recursive process. It bounds the variable $A$ to a fresh process name in the continuation $P$ and in any recursive instance of the thread $Q$. When unfolded, the definition receives some channel names which are bound to the variables $n_1,\dots,n_q$. Thus, $A,n_1,\dots,n_q$ are bound in $Q$ and $A$ is bound in $P$. A thread $A^p\langle\overline{m}\rangle$ allows for unfolding the recursive definition that is bound to the process variable $A$. The name restriction $(\nu^c n)P$ opens a fresh channel and bounds the variable $n$ to its name in the continuation $P$. The thread ${}^a[P]$ creates a new ambient labeled with $a$ and launches the continuation $P$ inside it. The thread $\Sigma^l_{i\in I}\pi_i.P_i$ denotes an external communication choice: it may compute the communication action $\pi_{i_0}$ and may launch the continuation $P_{i_0}$ if the dual communication action is available in the context at the correct location. A message reception bounds a variable to the name that is passed during the communication. Thus, in the thread $\$n?\{m\}.P$,

| | | | |
|---|---|---|---|
| $m,n,p$ | $\in$ | $\mathscr{N}_c$ | (channel name) |
| $A$ | $\in$ | $\mathscr{N}_p$ | (recursive variable name) |
| $p$ | $\in$ | $\mathscr{L}_{\mathrm{p}}$ | (program point label) |
| $c,d,e$ | $\in$ | $\mathscr{L}_{\mathrm{n}}$ | (channel name label) |
| $a,b$ | $\in$ | $\mathscr{L}_{amb}$ | (ambient label) |
| $\alpha$ | $\in$ | $\mathscr{L}_p$ | (recursive variable name label) |
| $I$ | $\in$ | $\{[\![1;n]\!] \mid n \in \mathbb{N}^*\}$ | (integer interval) |

| | | | |
|---|---|---|---|
| $P,Q$ | ::= | $\mathrm{let}^p\, A^\alpha\langle m_1,\dots,m_p\rangle = Q \text{ in } P$ | (recursive definition) |
| | $\mid$ | $A^p\langle n_1,\dots,n_p\rangle$ | (recursion unfolder) |
| | $\mid$ | $(\nu^c n)P$ | (restriction) |
| | $\mid$ | $\mathbf{0}$ | (inactivity) |
| | $\mid$ | $P \mid Q$ | (composition) |
| | $\mid$ | ${}^a[P]$ | (ambient) |
| | $\mid$ | $\Sigma^p_{i\in I}\pi_i.P_i$ | (communication choice) |
| | $\mid$ | $\Sigma^p_{i\in I}M_i.P_i$ | (capability choice) |

| | | | |
|---|---|---|---|
| $M$ | ::= | *enter n* | (synchronous entry) |
| | $\mid$ | *accept n* | (synchronous accept) |
| | $\mid$ | *exit n* | (synchronous exit) |
| | $\mid$ | *expel n* | (synchronous expel) |
| | $\mid$ | *merge+ n* | (positive synchronous merge) |
| | $\mid$ | *merge- n* | (negative synchronous merge) |

| | | | |
|---|---|---|---|
| $\pi$ | ::= | $\$n!\{p\}$ | (output action) |
| | $\mid$ | $\$n?\{m\}$ | (input action) |

| | | | |
|---|---|---|---|
| $ | ::= | *local* | (intra-ambient) |
| | $\mid$ | *s2s* | (inter-siblings) |
| | $\mid$ | *p2c* | (parent to child) |
| | $\mid$ | *c2p* | (child to parent) |

Figure 5.32: Syntax of BIO-*ambients*.

the variable $m$ is bound in $P$. There are four kinds of channeled communications. Local communications ($local\ n?\{m\}$/$local\ n!\{p\}$) happen between threads in the same ambient. Sibling communications ($sibling\ n?\{m\}$/$sibling\ n!\{p\}$) happen between threads that are located in sibling ambients. Parent to child communications ($c2p\ n?\{m\}$/$p2c\ n!\{p\}$) happen when a name sender is concurrent with an ambient that contains a name reception. Conversely, child to parent communications ($p2c\ n?\{m\}$/$c2p\ n!\{p\}$) happen when a reception is concurrent with an ambient that contains a name sending. The thread $\Sigma^{l}_{i\in I}M_i.P_i$ is an external choice between several capabilities. When two ambients interact, each of them must exhibit the good capability. They change the hierarchy of ambients, and launch their corresponding continuations. Capabilities $enter\ n/accept\ n$ control ambient entering, capabilities $exit\ n/expel\ n$ control ambient expelling and capabilities $merge{+}\ n/merge{-}\ n$ control ambient merging. It is worth noting that ambient merging is not symmetric. The ambient that computes the *merge-* capability is dissolved, and its content migrates inside the ambient that computes the *merge+*. Since ambients are not named, we use channel names to control the computation of these capabilities.

Usual rules about scope, substitution, and $\alpha$-conversion apply. We denote by $fn(P)$ the set of the names that are free in $P$, i.e names that are not under the scope of a binder, and by $bn(P)$ the set of the names that are bound in $P$.

## 5.5.2 Semantics

The system state is given by a pair $(D,P)$ where $P$ is a system and $D$ is a set of definitions of the form $A^{\alpha}\langle\overline{n}\rangle \triangleright^{p} P$ defined over distinct recursive variable names. The set of the recursive variable names that are defined in a set of definitions is denoted by $dn(D)$.

As usual, the operational semantics is given by both a transition relation and a congruence relation. The transition relation in Fig. 5.34 describes thread computation, whereas the congruence relation in Fig. 5.33 explicitly allows for thread interactions.

## 5.5.3 Non standard semantics

### 5.5.3.1 Partial interaction names

We define the set of partial interaction names as follows:

$$\begin{array}{c}\{enter;accept;expel;exit;merge{+};merge{-};mov\text{-}ambient;dis\text{-}ambient\}\\ \cup\{\$io \mid \$\in\{local;s2s;p2c;c2p\}, io\in\{?;!\}\}\\ \cup\{def_n \mid def\in\{rec;unfold\},\ n\in\mathbb{N}\}.\end{array}$$

$$\begin{array}{rcll}
P \mid Q & \equiv & Q \mid P & \\
(P \mid Q) \mid R & \equiv & P \mid (Q \mid R) & \\
P \mid \mathbf{0} & \equiv & P & \\
(\nu^c n)\mathbf{0} & \equiv & \mathbf{0} & \\
(\nu^c n)(\nu^d m)P & \equiv & (\nu^d m)(\nu^c n)P & \text{if } n \neq m \\
(\nu^c n)(P \mid Q) & \equiv & P \mid ((\nu^c n)Q) & \text{if } n \notin fn(P) \\
(\nu^c n)({}^a[P]) & \equiv & {}^a[(\nu^c n)P] & \\
(\nu^c n)P & \equiv & (\nu^c m)P[n \leftarrow m] & \text{if } m \notin fn(P) \\
\text{let}^p\, A^\alpha\langle\overline{m}\rangle = Q \text{ in } P & \equiv & (\text{let}^p\, A^\alpha\langle\overline{m}\rangle = Q \text{ in } P)\,[A \leftarrow B] & \text{if } B \notin fn(P \mid Q)
\end{array}$$

Figure 5.33: Congruence relation for BIO-*ambients*.

Their types and their arities are given as follows:

$$Ari = \begin{cases}
enter & \mapsto (2,0) \\
accept & \mapsto (2,0) \\
expel & \mapsto (2,0) \\
exit & \mapsto (2,0) \\
merge+ & \mapsto (2,0) \\
merge\text{-} & \mapsto (2,0) \\
rec_n & \mapsto (1,n+1) \\
unfold_n & \mapsto (n+2,0) \\
\$? & \mapsto (2,1) \\
\$! & \mapsto (3,0) \\
mov\text{-}ambient & \mapsto (1,1) \\
dis\text{-}ambient & \mapsto (1,0);
\end{cases}
\qquad
type = \begin{cases}
enter & \mapsto computation \\
accept & \mapsto computation \\
expel & \mapsto computation \\
exit & \mapsto computation \\
merge+ & \mapsto computation \\
merge\text{-} & \mapsto computation \\
rec_n & \mapsto replication \\
unfold_n & \mapsto computation \\
\$? & \mapsto computation \\
\$! & \mapsto computation \\
mov\text{-}ambient & \mapsto migration \\
dis\text{-}ambient & \mapsto computation
\end{cases}$$

Partial interaction names *enter*, *accept*, *expel*, *exit*, *merge+* and *merge-* denote threads that may compute a corresponding partial interaction. Such partial interactions are regular thread computations, they all have two parameters: the first one is the location of the thread and the second one denotes the name with which the capability applies. These partial interactions bound no variable.

The partial interaction name $rec_n$ denotes a recursive definition that requires an $n$-length message to be unfolded. Such a partial interaction is a replication, it has one parameter which is the name which is used to define the recursion. The location of the thread is not relevant, since the recursive instance will be launched at the unfolder thread location. This partial interaction bounds $n+1$ variables. The first variable encodes the location where the recursive instance is launched and the last $n$ variables are bound to the passed names. The partial interaction

$$\dfrac{M_{i_0} = \mathit{enter}\ n,\ M'_{j_0} = \mathit{accept}\ n}{\Big(D, {}^{a}[\Sigma_i^l M_i.P_i \mid Q] \mid {}^{a'}[\Sigma_j^{l'} M'_j.P'_j \mid Q']\Big) \xrightarrow{in(l,l')} \Big(D, {}^{a'}[P'_{j_0} \mid Q' \mid {}^{a}[P_{i_0} \mid Q]]\Big)}$$

$$\dfrac{M_{i_0} = \mathit{exit}\ n,\ M'_{j_0} = \mathit{expel}\ n}{\Big(D, {}^{a'}[{}^{a}[\Sigma_i^l M_i.P_i \mid Q] \mid \Sigma_j^l M'_j.P'_j \mid Q']\Big) \xrightarrow{out(l,l')} \Big(D, {}^{a}[P_{i_0} \mid Q] \mid {}^{a'}[P'_{j_0} \mid Q']\Big)}$$

$$\dfrac{M_{i_0} = \mathit{merge+}\ n,\ M'_{j_0} = \mathit{merge\text{-}}\ n}{\Big(D, {}^{a}[\Sigma_i^l M_i.P_i \mid Q] \mid {}^{a'}[\Sigma_j^{l'} M'_j.P'_j \mid Q']\Big) \xrightarrow{merge(l,l')} \big(D, {}^{a}[P'_{j_0} \mid Q' \mid P_{i_0} \mid Q]\big)}$$

$$\dfrac{\pi_{i_0} = \mathit{local}\ n?\{m\},\ \pi'_{j_0} = \mathit{local}\ n!\{p\}}{\Big(D, \Sigma_i^l \pi_i.P_i \mid \Sigma_j^{l'} \pi'_j.P'_j \mid Q\Big) \xrightarrow{local(l,l')} \big(D, P_{i_0}[m \leftarrow p] \mid P'_{j_0} \mid Q\big)}$$

$$\dfrac{\pi_{i_0} = \mathit{c2p}\ n?\{m\},\ \pi'_{j_0} = \mathit{p2c}\ n!\{p\}}{\Big(D, {}^{a}[\Sigma_i^l \pi_i.P_i \mid Q] \mid \Sigma_j^{l'} \pi'_j.P_j \mid Q'\Big) \xrightarrow{p2c(l,l')} \big(D, {}^{a}[P_{i_0}[m \leftarrow p] \mid Q] \mid P'_{j_0} \mid Q'\big)}$$

$$\dfrac{\pi_{i_0} = \mathit{p2c}\ n?\{m\},\ \pi'_{j_0} = \mathit{c2p}\ n!\{p\}}{\Big(D, \Sigma_i^l \pi_i.P_i \mid Q \mid {}^{a}[\Sigma_j^{l'} \pi'_j.P'_j \mid Q']\Big) \xrightarrow{2pc(l,l')} \big(D, P_{i_0}[m \leftarrow p] \mid Q \mid {}^{a}[P'_{j_0} \mid Q']\big)}$$

$$\dfrac{\pi_{i_0} = \mathit{s2s}\ n?\{m\},\ \pi'_{j_0} = \mathit{s2s}\ n!\{p\}}{\Big(D, {}^{a}[\Sigma_i^l \pi_i.P_i \mid Q] \mid {}^{a'}[\Sigma_j^{l'} \pi'_j.P'_j \mid Q']\Big) \xrightarrow{s2s(l,l')} \Big(D, {}^{a}[P_{i_0}[m \leftarrow p] \mid Q] \mid {}^{a'}[P'_{j_0} \mid Q']\Big)}$$

$$\dfrac{A \notin dn(D)}{(D, \mathrm{let}^p\ A^{\alpha}\langle \overline{m} \rangle = Q \ \mathrm{in}\ P) \xrightarrow{\varepsilon} (D \cup \{A^{\alpha}\langle \overline{m} \rangle \triangleright^p Q\}, P)}$$

$$\dfrac{A^{\alpha}\langle \overline{m} \rangle \triangleright^p Q \in D}{\Big(D, A^{p'}\langle \overline{n} \rangle\Big) \xrightarrow{fetch(p,p')} (D, Q[\overline{m} \leftarrow \overline{n}])}$$

$$\dfrac{(D,P) \xrightarrow{\lambda} (E,Q)}{(D, {}^{a}[P]) \xrightarrow{\lambda} (E, {}^{a}[Q])} \qquad \dfrac{(D,P) \xrightarrow{\lambda} (E,Q)}{(D, (\nu^c n)P) \xrightarrow{\lambda} (E, (\nu^c n)Q)}$$

$$\dfrac{(D,P) \xrightarrow{\lambda} (E,Q)}{(D, P \mid R) \xrightarrow{\lambda} (E, Q \mid R)} \qquad \dfrac{P' \equiv P,\ (D,P) \xrightarrow{\lambda} (E,Q),\ Q \equiv Q'}{\big(D, P'\big) \xrightarrow{\lambda} \big(E, Q'\big)}$$

Figure 5.34: Reduction relation for BIO-*ambients*.

name $unfold_n$ denotes a recursion unfolding. Such a partial interaction is a regular computation, it has $n+2$ parameters which are the location of the thread, the name of the definition which is unfolded, and the $n$ passed names. It bounds no variable.

Partial interaction names *local*?, *p2c*?, *c2p*?, and *s2s*? denote threads that are waiting for a name via a channel. These partial interactions have two parameters: the location of the thread that is computing the partial interaction and the name of the channel at which the thread listens. They bound one variable. Conversely, partial interaction names *local*!, *c2p*!, *p2c*!, and *s2s*! denote threads that are sending a name on a channel. These partial interactions have three parameters: the location of the thread that is computing the partial interaction, the name of the channel on which the name is sent, and the name that is passed. They bound no variable.

As in the case of the initial ambient calculus, ambients may migrate and may be dissolved. In order to compute the result of a migration or of a dissolution, the ambient must check some assumptions about its identity and about its location. In the case of a migration, it redefines its location and in the case of a dissolution, it replaces each occurrence of its identity with the location of its sibling ambient. Thus, we introduce two other partial interaction names: The name *mov-ambient* denotes the capability of migrating. Such a partial interaction has one parameter: the location of the ambient before the migration. It bounds one variable which is the updated location after the migration. The name *dis-ambient* denotes the capability of being dissolved. Such a partial interaction has one parameter: the location of the ambient.

#### 5.5.3.2 Formal rules

We now define the formal rules that implement migrations, dissolutions, and local communications. The rule *enter* which is described in Fig. 5.35 implements the migration of an ambient inside its parent ambient. It requires four threads: the first one is the moving ambient (*mov-ambient*), the second one is the target ambient (*mov-ambient*), the third one denotes a thread (*enter*) that provides to the first ambient the capability to move into the second one, and the fourth one provides the target ambient the capability to accept (*accept*) the moving ambient. The formal variables $X_1^1$, $X_1^2$, $X_1^3$, and $X_1^4$ respectively denote the locations of the migrating ambient, of the target ambient, of the entering thread, and of the accepting thread. The formal variables $X_2^3$ and $X_2^4$ denote respectively the name with which the entering capability applies and the name with which the accepting capability applies. The formal variables $I^1$ and $I^2$ respectively denote the identity of the moving ambient and of the target ambient. The migration is enabled if and only if:

1. the moving and the target ambients are located in the same ambient (i.e. $X_1^1 = X_1^2$);

$$enter = (4, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto \textit{mov-ambient} \\ 2 \mapsto \textit{mov-ambient} \\ 3 \mapsto \textit{enter} \\ 4 \mapsto \textit{accept}; \end{cases}$

2. $synchronization = \{X_1^1 = X_1^2; X_1^3 = I^1; X_1^4 = I^2; X_2^3 = X_2^4\}$;

3. $communication = [Y_1^1 \leftarrow I^2, Y_1^2 \leftarrow X_1^2]$;

4. $global = \emptyset$.

Figure 5.35: Formal rule for enter movement.

2. the moving ambient contains the capability to enter (i.e. $X_1^3 = I^1$);

3. the target ambient contains the capability to accept (i.e. $X_1^4 = I^2$);

4. the capabilities apply with the same name (i.e. $X_2^3 = X_2^4$).

At last, when interacting, the moving ambient updates its location which is now the identity of the target ambient. The formal variable $Y_1^1$ denotes the new location of the moving ambient. Thus, the variable $Y_1^1$ is associated with the variable $I^2$. The second ambient does not move. The formal variable $Y_1^2$ denotes the location of the second ambient. Thus, the variable $Y_1^2$ is associated with the variable $X_1^2$. There is no broadcast communication.

The rule *expel* which is described in Fig. 5.36 implements the migration of an ambient outside its parent ambient. It requires four threads: the first one is the expelled ambient (*mov-ambient*), the second one is the exited ambient (*mov-ambient*), the third one denotes a thread (*exit*) that provides to the first ambient the capability to exit the second one and the fourth one denotes a thread (*expel*) that provides the capability to the second ambient to expel the first one. The formal variables $X_1^1$, $X_1^2$, $X_1^3$, and $X_1^4$ respectively denote the locations of the expelled ambient, of the exited ambient, of the exiting capability, and of the expelling capability. The formal variables $X_2^3$ and $X_2^4$ respectively denote the name with which the exiting capability applies and the name with which the expelling capability applies. The formal variables $I^1$ and $I^2$ respectively denote the identity of the exiting ambient and of the expelling ambient. The migration is enabled if and only if:

1. the expelling ambient contains the exiting ambient (i.e. $X_1^1 = I_2$);

$$expel = (4, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto \textit{mov-ambient} \\ 2 \mapsto \textit{mov-ambient} \\ 3 \mapsto \textit{exit}; \\ 4 \mapsto \textit{expel} \end{cases}$

2. $synchronization = \{X_1^1 = I^2; X_1^3 = I^1; X_1^3 = I^2; X_2^3 = X_2^4;\}$;

3. $communication = [Y_1^1 \leftarrow X_1^2, Y_1^2 \leftarrow X_1^2]$;

4. $global = \emptyset$.

Figure 5.36: Formal rule for expelling.

2. the exiting ambient contains the capability to exit (i.e. $X_1^3 = I^1$);

3. the expelling ambient contains the capability to expel (i.e. $X_1^4 = I^2$);

4. the capabilities apply with the same name (i.e. $X_2^3 = X_2^4$).

At last, when interacting, the exiting ambient updates its location which is now the location of the expelling ambient. The formal variable $Y_1^1$ denotes the new location of the exiting ambient. Thus, the variable $Y_1^1$ is associated with the variable $X_1^2$. The second ambient does not move. The formal variable $Y_1^2$ denotes the location of the second ambient. Thus, the variable $Y_1^2$ is associated with the variable $X_1^2$. There is no broadcast communication.

The formal rule *merge* which is described in Fig. 5.37 implements the merging of two ambients. It requires four threads: the first two ones denote the merging ambients and the last two ones denote some threads. The first ambient will remain unchanged (*mov-ambient*) and the second will be dissolved (*dis-ambient*). Threads compute merging partial interactions (*merge+* and *merge-*). The formal variables $X_1^1$, $X_1^2$, $X_1^3$, and $X_1^4$ respectively denote the locations of the two ambients and the locations of the two threads. The formal variables $X_2^3$ and $X_2^4$ denote the name with which the two threads capabilities apply. The formal variables $I^1$ and $I^2$ encode the two ambients identities. The ambient merging is enabled if and only if:

1. the ambients are located inside the same location (i.e. $X_1^1 = X_1^2$);

2. each thread is located in the correct ambient (i.e. $X_1^3 = I^1$ and $X_1^4 = I^2$);

$$\mathit{merge} = (4, \mathit{component}, \mathit{synchronization}, \mathit{communication}, \mathit{global})$$

where

1. $\mathit{component} = \begin{cases} 1 \mapsto \mathit{mov\text{-}ambient} \\ 2 \mapsto \mathit{dis\text{-}ambient} \\ 3 \mapsto \mathit{merge+} \\ 4 \mapsto \mathit{merge\text{-}} \end{cases}$

2. $\mathit{synchronization} = \{X_1^1 = X_1^2; X_1^3 = X_1^1; X_1^4 = X_1^2; X_2^3 = X_2^4\}$;

3. $\mathit{communication} = [Y_1^1 \leftarrow X_1^1]$;

4. $\mathit{global} = [I^2 \mapsto I^1]$.

Figure 5.37: Formal rule for merging.

3. the two capabilities apply with the same name (i.e. $X_2^3 = X_2^4$);

The first ambient does not move. The formal variable $Y_1^1$ denotes the location of the first ambient. Thus, the variable $Y_1^1$ is associated with the variable $X_1^1$. Any thread in the second ambient is now in the first one. Thus, any instance of the second ambient identity in the whole system must be updated with the first ambient identity $[I^2 \mapsto I^1]$.

The rule *local-com* in Fig. 5.38 implements local communications. A local

$$\mathit{local\text{-}com} = (2, \mathit{component}, \mathit{synchronization}, \mathit{communication}, \mathit{global})$$

where

1. $\mathit{component} = \begin{cases} 1 \mapsto \mathit{local}? \\ 2 \mapsto \mathit{local}! \end{cases}$

2. $\mathit{synchronization} = \{X_1^1 = X_1^2; X_2^1 = X_2^2\}$;

3. $\mathit{communication} = [Y_1^1 \leftarrow X_3^2]$;

4. $\mathit{global} = \emptyset$.

Figure 5.38: Formal rule for local communication.

communication requires two threads that denote the two communicating threads (*local*? and *local*!). The formal variables $X_1^1$ and $X_1^2$ denote the locations of these two threads. The formal variables $X_2^1$ and $X_2^2$ denote the name on which the communication is performed. The local communication is enabled if and only if:

1. the threads are located in the same ambient (i.e. $X_1^1 = X_1^2$);
2. the threads communicate through the same channel (i.e. $X_2^1 = X_2^2$);

The formal variable $X_3^2$ denotes the name which is sent. In the case when the communication is enabled, a new variable is declared and associated with the output channel name. The new variable is denoted by the formal variable $Y_1^1$. Name passing is described by associating the formal variable $Y_1^1$ with the formal variable $X_3^2$. These is no broadcast communication.

The rule *p2c* in Fig. 5.39 implements a communication from a parent to a child. Such a communication requires three threads that denote the two com-

$$p2c = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto c2p? \\ 2 \mapsto p2c! \\ 3 \mapsto mov\text{-}ambient \end{cases}$
2. $synchronization = \{X_1^1 = I^3; X_1^2 = X_1^3; X_2^1 = X_2^2\}$;
3. $communication = [Y_1^1 \leftarrow X_3^2, Y_1^3 \leftarrow X_1^3]$;
4. $global = \emptyset$.

Figure 5.39: Formal rule for parent to child communication.

municating threads (*c2p*? and *p2c*!) and an ambient (*mov-ambient*). The formal variables $X_1^1$ and $X_1^2$ denote the location of the two communicating threads and the variable $X_1^3$ denotes the location of the ambient. The formal variables $X_2^1$ and $X_2^2$ denote the name on which the communication is performed. The parent to child communication is enabled if and only if:

1. the ambient contains the thread that is waiting for a message (i.e. $X_1^1 = I^3$);
2. the thread that sends the message is at the same location as the ambient (i.e. $X_1^2 = X_1^3$);
3. the two threads communicate over the same channel (i.e. $X_2^1 = X_2^2$).

The formal variable $X_3^2$ denotes the name which is sent. In the case when the communication is enabled, a new variable is declared and associated with the output channel name. This new variable is denoted by the formal variable $Y_1^1$. Name passing is described by associating the formal variable $Y_1^1$ with the formal variable $X_3^2$. The ambient does not move. The formal variable $Y_1^3$ denotes the location of the ambient. Thus, the variable $Y_1^3$ is associated with the variable $X_1^3$.

The rule *c2p* in Fig. 5.40 implements a communication from a child to a parent. Such a communication requires three threads that denote the two communicating

$$c2p = (3, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto p2c? \\ 2 \mapsto c2p! \\ 3 \mapsto mov\text{-}ambient \end{cases}$
2. $synchronization = \{X_1^2 = I^3; X_1^1 = X_1^3; X_2^1 = X_2^2\}$;
3. $communication = [Y_1^1 \leftarrow X_3^2, Y_1^3 \leftarrow X_1^3]$;
4. $global = \emptyset$.

Figure 5.40: Formal rule for content to parent communication.

threads (*p2c*? and *c2p*!) and an ambient (*mov-ambient*). The formal variables $X_1^1$ and $X_1^2$ denote the location of the two communicating threads and the variable $X_1^3$ denotes the location of the ambient. The formal variables $X_2^1$ and $X_2^2$ denote the name on which the communication is performed. The child to parent communication is enabled if and only if:

1. the ambient and the thread that is waiting for a message are at the same location (i.e. $X_1^1 = X_1^3$);
2. the ambient contains the thread that sends the message (i.e. $X_1^2 = I^3$);
3. the two threads communicate over the same channel (i.e. $X_2^1 = X_2^2$).

The formal variable $X_3^2$ denotes the name which is sent. In the case where the communication is enabled, a new variable is declared and associated with the output channel name. This new variable is denoted by the formal variable $Y_1^1$. Name passing is described by associating the formal variable $Y_1^1$ with the formal variable $X_3^2$. The ambient does not move. The formal variable $Y_1^3$ denotes the location of the ambient. Thus, the variable $Y_1^3$ is associated with the variable $X_1^3$.

The rule *sibling-com* in Fig. 5.41 implements a communication through two sibling ambients. Such a communication requires four threads that denote the two communicating threads (*s2s*? and *s2s*!) and the two sibling ambients (*mov-ambient*). The formal variables $X_1^1$ and $X_1^2$ denote the locations of the two

$$\textit{sibling-com} = (4, \textit{component}, \textit{synchronization}, \textit{communication}, \textit{global})$$

where

1. $\textit{component} = \begin{cases} 1 \mapsto \textit{s2s}? \\ 2 \mapsto \textit{s2s}! \\ 3 \mapsto \textit{mov-ambient} \\ 4 \mapsto \textit{mov-ambient} \end{cases}$

2. $\textit{synchronization} = \{X_1^1 = I^3; X_1^2 = I^4; X_1^3 = X_1^4; X_2^1 = X_2^2\};$

3. $\textit{communication} = [Y_1^1 \leftarrow X_3^2, Y_1^3 \leftarrow X_1^3, Y_1^4 \leftarrow X_1^4];$

4. $\textit{global} = \emptyset.$

Figure 5.41: Formal rule for sibling communication.

communicating threads and the variables $X_1^3$ and $X_1^4$ denote the ambient locations. The formal variables $X_2^1$ and $X_2^2$ denote the name on which the communication is performed. The communication between sibling ambients is enabled if and only if:

1. the first ambient contains the thread that is waiting for a message (i.e. $X_1^1 = I^3$);

2. the second ambient contains the thread that sends the message (i.e. $X_1^2 = I^4$);

3. the two ambients have the same location (i.e. $X_1^3 = X_1^4$);

4. the two threads communicate over the same channel (i.e. $X_2^1 = X_2^2$).

The formal variable $X_3^2$ denotes the name which is sent. In the case when the communication is enabled, a new variable is declared and associated with the output channel name. This new variable is denoted by the formal variable $Y_1^1$. Name passing is described by associating the formal variable $Y_1^1$ with the formal variable $X_3^2$. The two ambients do not move. The formal variables $Y_1^3$ and $Y_1^4$ denote the locations of these ambients. Thus, the variable $Y_1^3$ is associated with the variable $X_1^3$ and the variable $Y_1^4$ is associated with the variable $X_1^4$.

The rule $\textit{rec}_n$ in Fig. 5.42 implements recursion unfolding. The parameter $n$ denotes the number of arguments. Recursion unfolding requires two threads. The first one is a recursive definition ($\textit{rec}_n$) that requires $n$ parameters and the second one it an unfolding thread ($\textit{unfold}_n$) that passes $n$ parameters. The formal variable $X_1^2$ denotes the location of the unfolding thread. We recall that definitions are in the ether, so they have no location. The formal variable $X_1^1$ denotes the

$$rec_n = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto rec_n \\ 2 \mapsto unfold_n \end{cases}$

2. $synchronization = \{X_1^1 = X_2^2\};$

3. $communication = [Y_1^1 \mapsto X_1^2, Y_{1+i}^1 \mapsto X_{2+i}^2];$

4. $global = \emptyset.$

Figure 5.42: Formal rule for unfolding recursions.

variable name that has been used at the creation of the definition and the variable $X_2^2$ denotes the variable name that is used by the unfolding thread. The recursion may be unfolded if and only if these two variable names are the same $(X_1^1 = X_2^2)$. In such a case, the recursion is unfolded at the unfolding thread location, that is described by the association $Y_1^1 \mapsto X_1^2$ where the formal variable $Y_1^1$ denotes the location of the unfolded instance. Moreover, $n$ variables are created and associated with the passed parameters. This is described by the associations $Y_{i+1}^1 \mapsto X_{i+2}^2$ where each variable $Y_{i+1}^1$ denotes the $i$-th created variable and each variable $X_{i+2}^2$ denotes the $i$-th passed name. These is no broadcast communication.

#### 5.5.3.3 Abstract syntax extraction

We now define the syntax extraction function that takes a program describing the initial state of a mobile system in the standard syntax and extract the abstract syntax of it.

Program points are ambient creation, actions, and definitions. The abstract syntax maps each program point label $l$ to the following set of partial interactions according to the syntax of the program point:

- the label $a$ of a program point ${}^a[P]$ is associated with the interface $\{loc\}$ and the following set of partial interactions:

$$\{(mov\text{-}ambient, [loc], [loc], \emptyset, \{\{(a, \emptyset)\}\}); (dis\text{-}ambient, [loc], [], \emptyset, \{\emptyset\})\}.$$

- the label $p$ of a program point $P = \Sigma_{i \in I}^{p} \pi_i.P_i$ is associated with the interface $\{loc\} \cup fn(P)$ and with the following set of partial interactions:

$$\{(act(\pi_i), arg(\pi_i), bound(\pi_i), \emptyset, \{\beta(P_i, \emptyset)\}) \mid i \in I\},$$

where $act = \begin{cases} \$n?\{m\} \mapsto \$? \\ \$n!\{p\} \mapsto \$!, \end{cases}$ $arg = \begin{cases} \$n?\{m\} \mapsto [loc;n] \\ \$n!\{p\} \mapsto [loc;n;p], \end{cases}$

and $bound = \begin{cases} \$n?\{m\} \mapsto [m] \\ \$n!\{p\} \mapsto []. \end{cases}$

- the label $p$ of a program point $P = \Sigma^p_{i\in I} M_i.P_i$ is associated with the interface $\{loc\} \cup fn(P)$ and with the following set of partial interactions:

$$\{(act(M_i), [loc; name(M_i)], [], \emptyset, \{\beta(P_i, \emptyset)\}) \mid i \in I\},$$

where $act = \begin{cases} enter\ n & \mapsto enter \\ accept\ n & \mapsto accept \\ exit\ n & \mapsto exit \\ expel\ n & \mapsto expel \\ merge\text{+}\ n & \mapsto merge\text{+} \\ merge\text{-}\ n & \mapsto merge\text{-}, \end{cases}$ and $name = \begin{cases} enter\ n & \mapsto n \\ accept\ n & \mapsto n \\ exit\ n & \mapsto n \\ expel\ n & \mapsto n \\ merge\text{+}\ n & \mapsto n \\ merge\text{-}\ n & \mapsto n. \end{cases}$

- the label $p$ of a program point $A^p\langle p_1, \ldots, p_n\rangle$ is associated with the interface $\{loc; A\} \cup \{p_1; \ldots; p_n\}$ and with the following set of partial interactions:

$$\{unfold_n, [loc; A; p_1; \ldots; p_n], [], \emptyset, \{\emptyset\}\},$$

- the label $p$ of a program point $\mathrm{let}^p\ A^\alpha\langle m_1, \ldots, m_n\rangle = Q \text{ in } P$ is associated with the interface $\{A\} \cup (fn(Q) \setminus \{m_1; \ldots; m_n\})$ and with the following set of partial interactions:

$$\{rec_n, [A], [m_1; \ldots; m_n], \emptyset, \beta(Q, \emptyset)\}.$$

Continuation computations use an auxiliary primitive $\beta$. This primitive computes the continuation[4]. It is defined inductively over the standard syntax of the continuation, as follows:

$$\begin{array}{rcl} \beta(\mathrm{let}^p\ A^\alpha\langle m_1, \ldots, m_p\rangle = Q \text{ in } P, E_s) & = & \beta(P, E_s) \cup \{(p, E_s[A \mapsto \alpha])\} \\ \beta(A^p\langle \overline{m}\rangle, E_s) & = & \{(A^p\langle \overline{m}\rangle, E_s)\} \\ \beta((\nu^c n)P, E_s) & = & \beta(P, E_s[n \mapsto c]) \\ \beta(\mathbf{0}, E_s) & = & \emptyset \\ \beta(P \mid Q, E_s) & = & \beta(P, E_s) \cup \beta(Q, E_s) \\ \beta({}^a[P], E_s) & = & \{(a, E_s)\} \cup \beta(P, E_s[loc \mapsto a]) \\ \beta(\Sigma^p_i \pi_i.P_i, E_s) & = & \{(\Sigma^p_i \pi_i.P_i, E_s)\} \\ \beta(\Sigma^p_i M_i.P_i, E_s) & = & \{(\Sigma^p_i \pi_i.P_i, E_s)\} \end{array}$$

[4]We avoid defi ning a set of continuations, since we use no non-deterministic choice.

The set of initial states is defined as follows:

$$\beta(\mathscr{S}, [loc \mapsto (\text{top}, \varepsilon)]),$$

where the location $(\text{top}, \varepsilon)$ denotes the address of the top level ambient.

### 5.5.4 Correspondence

**Proposition 5.5.1.** *For each system, the extracted syntax satisfies the sufficient properties in Def. 4.5.11. Thus, allocated markers are unambiguous.*

The correspondence between the standard and the non standard semantics is established as follows:

**Theorem 5.5.2.** *The standard and the non standard semantics are in weak bisimulation.*

## 5.6 Extending the framework

Our meta language cannot model the bang operator, since spontaneous replication prevents from tracking the history of the threads. So bang operator must be simulated by using fetched replication. Moreover, we cannot deal with an equational theory such as in the applied-$\pi$ calculus [1], or with symmetric communications such as in the solo-calculus [49, 48] or in the fusion-calculus [65], since this feature may destroy the origin of the values that are used in the system. We have also avoided testing term equalities (in the *spi*-calculus we have assumed that we know the syntactic skeleton of the term we match). We do not show how we can deal with higher order communications [70] although our meta language can model them. We explain how we can relax some of these limits.

### 5.6.1 Testing term equalities

We have avoided testing full term equalities when encoding the *spi*-calculus, because term unification may require visiting an unbounded number of nodes in a term. Anyway, we may consider abstract version of equality test by using $k$-depth limiting. Let $M$ and $N$ be two terms. We say that $M$ and $N$ are 0-equal if and only if $M$ and $N$ are associated to a same name. Moreover, we say that $M$ and $N$ are 0-disequal if and only if either $M$ or $N$ is associated to a name and if $M$ and $N$ are not 0-equal. Then, we say that $M$ and $N$ are $n+1$-equal either if they are $n$-equal, or if there are associated with the same term constructor applied to pairwise $n$-equal arguments. Moreover, we say that $M$ and $N$ are $n+1$-disequal if they are *n*-disequal, if they are associated with two distinct term constructor, or if they are associated with the same term constructor with a pair of arguments that are *n*-disequal.

Then there are two possible approaches.

- The first approach consists in disallowing equality tests among incomparable terms (neither *k*-equal, nor *k*-disesqual) at run-time. So that we only consider type-safe computations. In the abstract, we over approximate the relation *k*-equal, by enumerating all potential patterns.

- The second approach consists in considering any equality test and any unification. In the abstract, we will soundly abstract the equality test by the negation of the relation *k*-disequal. This way, we will consider that two terms may be equal whenever the abstraction cannot prove that they are not equal.

## 5.6.2 Higher order model encoding

We have not investigate the encoding of higher order communication. Nevertheless, we believe that usual pointer-based encoding such as in [70], may give convenient result. But it is not obvious, since such an encoding may lose all the structure of the model, which is not the case for the models we have already encoded.

## 5.6.3 Encoding the projective brane calculus

Projective brane calculus [28] is a refinement of brane calculus [14], where membranes are oriented. Some capability may interact either with the outside, or with the inside of a membrane. This notion is relative to the point of view. Moreover, orientation dynamically changes during computation.

To model it, we use one program point at the meta language level per program point/orientation pair at the brane calculus level. Then, the encoding is straightforward.

# Chapter 6

# Context approximation

Until now, we have only considered closed systems. However, a mobile system is usually a small part of a bigger system. The rest of the bigger system is called the context. The system part under assumption is called an open system. The open system aims at interacting with its context. Nevertheless, this context may be known or not: It can be made of some trustful threads or of an hostile intruder. This context may either try to force the open system into violating its usual behaviors, or exploit a security leak in order to spy some sensitive information of the open system. The aim of this chapter is to extend our framework to open systems.

We now give the summary of this chapter. In Sect. 6.1, we recall usual methods for including a context in the semantics. In Sect. 6.2, we recall the framework that we proposed in [32] in order to deal with an arbitrary context in the $\pi$-calculus and establishes both the soundness and the completeness of the approach. In Sect. 6.3, we show how this context abstraction can be applied to the meta language level. In Sect. 6.3.4, we give incompleteness counter-examples in the case of the spi-calculus and in the case of the ambient-like calculi.

## 6.1 Introduction

### 6.1.1 Three approaches

There are different approaches to model the context.

#### 6.1.1.1 Modeling a context in the language

We can consider that the context is encoded in the concrete semantics by an element (i.e. a system part) of the model. This approach has been followed in [5, 6] in the case of the $\pi$-calculus and in [5, 4] in the case of the spi-calculus. The main advantage of this approach is that it can be applied straightforwardly by analyzing the parallel composition of the open system and its context. The main drawback is that we may give too less control of the system to the context. Thus, we may fail to discover some attacks.

#### 6.1.1.2 Abstracting a context

We can describe explicitly or implicitly an abstraction of the knowledge of the context and give some rules to model the interaction with the context and the open system. We must describe how the open system is modified when it interacts with the context and how the context knowledge is updated. For instance, the Dolev and Yao approach [31] is widely used when analyzing cryptographic protocols. The context (which is called the intruder in cryptographic protocols) may control each channel (to receive communicated terms and to send spoiling terms). Moreover, the context may build any term from the messages it may spy from the open system.

The main advantage of abstracting the context by a set of rules is that the power of the intruder may overpass the semantics of the model: the rules that specify the intruder's behavior can be chosen arbitrarily. We can capture attacks that are not described in the model, provided that we model them in the intruder abstraction. The mail drawback is that the abstraction may not be related to the semantics. Thus, it is arbitrary.

#### 6.1.1.3 Abstracting any context in the model

We can derive the semantics of open systems by abstraction of the semantics of closed systems. In [36, 32], we propose a semantics for open systems in the $\pi$-calculus. In this semantics, the behavior of the context is approximated: we only keep the set of the channel names with which the context may use. We have proved in [32] that this approach is both sound and complete with respect to an abstraction function that maps each computation trace for any closed system that encloses an open system into a computation trace of this open system. So given an open system, the abstraction of a trace of a closed system that encloses this open system is always a trace of the context independent semantics of the open system (*soundness*). Reciprocally, any trace in the context independent semantics is the abstraction of a trace of a closed system that encloses the open system (*completeness*). Our approach provides sound abstractions at the meta language level. But completeness is not provided in general: completeness relies heavily on the structure of the encoded model.

The main advantage of this approach is that we can relate the intruder power to the semantics of the model. The intruder may do whatever any context encoded in the model may do. This allows for modular analysis. We detect all the potential interactions of the system in any context that can be written in the model. Moreover, if the abstraction is proved complete, the intruder may not do more than it is allowed to in the semantics.

#### 6.1.1.4 Some remarks

*Remark* 6.1.1. In the case of the $\pi$-calculus, we usually design a label transition system (*LTS*) to describe the context behavior. The knowledge of the system is implicitly described as the set of the channel names that are not in the scope of a restriction. There are two kinds of transitions: the $\tau$ transitions denote some transitions inside the open system, some transitions allow the extrusion of open system channel names to the context, and some others allow the context to send some names through an untrusted channel (through channels the scope of the name of which is not restricted). The *LTS* semantics are widely used when proving *bisimulations*. The idea is to prove that two system parts have the same behavior (with respect to an observational semantics) when stimulated with the same sequences of interactions with a context. Our requirements are quite different. On one hand, we want the context independent semantics and the closed semantics to coincide on the trace level and on the other hand, we are only interested in reachability properties instead of relating observation semantics of two systems.

*Remark* 6.1.2. These three approaches enjoy strong connections. For instance, if the abstraction of a context behavior by a set of rules satisfies the semantics, then it may be described by a system in the model. Moreover the proof of completeness of the abstraction of any context is very likely to use the existence of a generic context that can be encoded in the model and that abstracts the behavior of any context. In such a case, the abstraction of any context boils down to modeling the context by a given system.

## 6.2 Context independent semantics for the $\pi$-calculus

In this section, we explain how we can extend the non standard semantics that we have proposed in Chap. 2 for closed mobile systems in the $\pi$-calculus to open systems. These results have been published in [32].

### 6.2.1 Context approximation

An open system $\mathscr{S}$ is a part of a bigger closed system, the rest of which is called its context. The context is a set of threads, concurrently running with $\mathscr{S}$. We represent this context by the set of channel names it shares with the system $\mathscr{S}$, we call such names the *unsafe names*, and approximate the behavior of the system $\mathscr{S}$ as if it was an intruder who was able to compose any possible thread working on these channel names. An interaction between the system $\mathscr{S}$ and its context may only consist in a communication between a thread $p_{\mathscr{S}}$ of the first and a thread $p_{\text{cont}}$ of the second, via a channel the name of which is unsafe. This communication is called *spying* if $p_{\text{cont}}$ is the receiver, and *spoiling* if $p_{\text{cont}}$ is the message sender. When spying, the context listens to obtain new channel names which become unsafe. When spoiling, the context may pass some names to $\mathscr{S}$. Each of these names is either an unsafe name denoting a channel opened by a binder of the system $\mathscr{S}$, or a name denoting a channel opened by the context itself; as a consequence, we have to introduce an infinite set of unsafe names for the channels that the context may have opened. Eventually, spoiling may lead to the replication of a resource, which requires the allocation of an unambiguous marker, otherwise the consistency of the semantics would not be preserved.

Since $\alpha$-conversion allows us to choose the names of the new channels opened by the context, we may assume that those channels have been declared by recursive instances of a single thread. By choosing some fresh distinct program point labels 0, $cont_?$, $cont_! \in \mathscr{L}$ and a name $ext \in \mathscr{N} \setminus bn(\mathscr{S})$, such channels will be seen as if they had been created by the restriction ($\nu$ *ext*) of a recursive instance of a thread the marker of which is $t_n$, where $t_n$ is recursively defined as follows:

$$\begin{cases} t_0 & = & N((cont_?, \text{cont}_!), \varepsilon, \varepsilon) \\ t_{n+1} & = & N((cont_?,\ cont_!), \varepsilon, t_n). \end{cases}$$

Thus, we choose the set of the names of the channels that are opened by the context as the set $\{(ext, t_n) \mid n \in \mathbb{N}\}$. We denote this set by *en*. We also assume that all spoiling messages are recursive instances of a single thread the first action of which is labeled with 0.

The coherence of our semantics mainly relies on the fact that during a computation sequence, there cannot be two different instances of a single thread with the same marker. We guarantee this property by associating to each spoiling message a fresh marker in the set $\{t_n \mid n \in \mathbb{N}\}$.

### 6.2.2 Open transition system

A non standard configuration is now a triple $(C, U, F)$, where $C$ is a set of threads, $U$ is a set of channel names, and $F$ is a set of markers. The set $C$ contains the running threads. The set $U$ contains all names $(a, id)$ such that the channel opened by the restriction ($\nu$ $a$) of the thread instance tagged with the marker *id* is unsafe. The set $F$ contains fresh markers which have not been used as markers for spoiling

$$\mathscr{C}_0^{\mathrm{o}}(\mathscr{S}) = \left\{ (Ct, en, \{t_n \mid n \in \mathbb{N}\}) \;\middle|\; \begin{array}{l} Ct \in \beta(\mathscr{S}, \varepsilon, E), \\ E \in (fn(\mathscr{S}) \to en) \end{array} \right\}$$

(a) Initial configurations.

$$\frac{C \xrightarrow{\lambda}_{\mathrm{e}} C'}{(C, U, F) \overset{\lambda}{\looparrowright} (C', U, F)}$$

(b) Internal interactions.

$$\frac{t = (x!^j[x_1, \ldots, x_n]P, id, E),\ E(x) \in U,\ Ct \in \beta(P, id, E)}{(C \cup \{t\}, U, F) \overset{(0,j)}{\looparrowright} (C \cup Ct, U \cup \{E(x_k) \mid k \in [\![1;n]\!]\}, F)}$$

(c) Spied communication.

$$\frac{\begin{cases} t = (y?^i[y_1, \ldots, y_n]P, id, E), \\ E(y) \in U,\ c_1, \ldots, c_n \in U, \\ Ct \in \beta(P, id, E[y_k \mapsto c_k]) \end{cases}}{(C \cup \{t\}, U, F) \overset{(i,0)}{\looparrowright} (C \cup Ct, U, F)}$$

(d) Spoiled communication.

$$\frac{\begin{cases} t = (*y?^i[y_1, \ldots, y_n]P, id, E), \\ E(y), c_1, \ldots, c_n \in U, \\ id_! \in F, \\ id_* = N((i,0), id, id_!), \\ Ct \in \beta(P, id_*, E[y_k \mapsto c_k]) \end{cases}}{(C \cup \{t\}, U, F) \overset{(i,0)}{\looparrowright} (C \cup \{t\} \cup Ct, U, F \setminus \{id_!\})}$$

(e) Spoiled resource replication.

Figure 6.1: Context independent non standard semantics.

messages. At the beginning of the system computation, free names have to be chosen among the set of initial unsafe names. We make no assumptions about the past of the system, so that distinct free names may be bound to the same unsafe name. This is especially useful, when analyzing an instance of a resource without any knowledge of the relations between channel names that have been communicated to it.

The transition relation $\looparrowright$ takes into account the computations inside the mobile system $\mathscr{S}$, as well as the computations involving the system $\mathscr{S}$ and its context. Initial non standard configurations and computation rules are given in Fig. 6.1. Their definition both use the definition of the efficient semantics for the closed mobile systems of the $\pi$-calculus that is given in Fig. 2.8 on page 31.

### 6.2.3 Coherence

We propose to establish a relation between the non standard semantics of closed and open systems. Let $\mathscr{S}_{\mathrm{I}}(x_1,\ldots,x_n)$ be an open system the set of the free names of which are exactly the set $\{x_i \mid i \in [\![1;n]\!]\}$. We want to construct a projection function $\Pi_\tau$, such that:

- any non standard computation sequence $\tau$ of a closed system of the form $(\nu\, c_1)\ldots(\nu\, c_k)(\mathscr{S}_{\mathrm{I}}(c_{i_1},\ldots,c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1},\ldots,c_{j_l}))$, is mapped to a non standard computation sequence $\Pi_\tau(\tau)$ of the open system $\mathscr{S}_I$;

- reciprocally for any non standard computation sequence $\tau'$ of the open system $\mathscr{S}_{\mathrm{I}}$, there exists a computation sequence $\tau$ of a closed system of the form $(\nu\, c_1)\ldots(\nu\, c_k)\ (\mathscr{S}_{\mathrm{I}}(c_{i_1},\ldots,c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1},...c_{j_l}))$, such that $\tau' = \Pi_\tau(\tau)$.

#### 6.2.3.1 Trace abstraction

Let $\mathscr{L}_{\mathrm{I}} \subseteq \mathscr{L}$ be the part of the labels occurring in $\mathscr{S}_{\mathrm{I}}$ and $\mathscr{N}_{\mathrm{I}} \subseteq \mathscr{N}$ be the part of the names used in name restrictions of $\mathscr{S}_{\mathrm{I}}$. The function *lab* maps each syntactic component beginning with an action to the label of this action. We introduce two one-to-one functions in order to interpret names and threads created by the context of $\mathscr{S}_{\mathrm{I}}$: let $\Phi_{\mathscr{M}}$ be a one-to-one map from the set $(\mathscr{L} \times \mathscr{M})$ into the set $\{t_n \mid n \in \mathbb{N}\}$, and $\Phi_{\mathscr{N}}$ be a one-to-one function from the set $(\mathscr{N} \times \mathscr{M})$ into the set *en*. We now define the projection $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})$ which transforms each computation sequence of a closed mobile system $\mathscr{S} = (\nu\, \overline{c})(\mathscr{S}_{\mathrm{I}}(c_{i_1},\ldots,c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1},\ldots,c_{j_l}))$ into a computation sequence of the part $\mathscr{S}_{\mathrm{I}}$ of the system $\mathscr{S}$. We assume without any loss of generality that $fn(\mathscr{S}_{\mathrm{I}}) = \{c_{i_k} \mid k \in [\![1;n]\!]\}$, and that no name occurs twice as an argument of a name binder.

We first project each syntactic component label, by replacing each label which does not occur in $\mathscr{S}_{\mathrm{I}}$ by the unique label of the context, that is to say the 0 label.

**Definition 6.2.1 (program point label projection).** The projection $\Pi_{\mathrm{l}}(l)$ of a syntactic component label $l \in \mathscr{L}$ is defined as follows:

$$\Pi_{\mathrm{l}}(l) = \begin{cases} l & \text{if } l \in \mathscr{L}_{\mathrm{I}} \\ 0 & \text{otherwise.} \end{cases}$$

Then, we apply the syntactic component label projection pair-wise on transition labels.

**Definition 6.2.2 (transition label projection).** The projection $\Pi_\lambda(i,j)$ of transition label $(i,j) \in \mathscr{L}^2$ is defined as follows:

$$\Pi_\lambda(i,j) = (\Pi_{\mathrm{l}}(i), \Pi_{\mathrm{l}}(j)).$$

Next, we project the instance markers of the syntactic components of $\mathscr{S}_{\mathrm{I}}$. Only the right sibling of such markers may be the marker of a syntactic component of the context, since the replicated resource necessarily belongs to $\mathscr{S}_{\mathrm{I}}$. When a resource is replicated by a message of the context, we replace its syntactic component, and compute a coherent marker according to $\Phi_{\mathscr{M}}$.

**Definition 6.2.3 (marker projection).** Marker projection is defined as follows:

$$\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}} : \begin{cases} N((i,j),t_1,t_2) & \mapsto N(\Pi_\lambda(i,j), \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(t_1), \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(t_2)) & \text{if } j \in \mathscr{L}_{\mathrm{I}} \\ N((i,j),t_1,t_2) & \mapsto N(\Pi_\lambda(i,j), \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(t_1), \Phi_{\mathscr{M}}(j,t_2)) & \text{if } j \notin \mathscr{L}_{\mathrm{I}} \\ \qquad \varepsilon & \mapsto \varepsilon. \end{cases}$$

We now project channel names. For a channel opened by a name restriction of $\mathscr{S}_{\mathrm{I}}$, we just project the marker. For those opened by the context, we replace the name restriction by the unique restriction $(\nu\ ext)$ of the context, and compute the coherent marker according to $\Phi_{\mathscr{N}}$.

**Definition 6.2.4.** Channel name projection is defined as follows:

$$\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(x, id_x) = \begin{cases} (x, \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_x)) & \text{if } x \in \mathscr{N}_{\mathrm{I}} \\ \Phi_{\mathscr{N}}(x, id_x) & \text{otherwise.} \end{cases}$$

We easily project an instance of a syntactic component of $\mathscr{S}_{\mathrm{I}}$ by projecting its marker and each channel name that occurs in its environment.

**Definition 6.2.5 (thread projection).** Thread projection is defined as follows:

$$\Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(P, id, E) = (P, \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id), [x \mapsto \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(x))]).$$

Then, we project a configuration by projecting all the threads the syntactic component of which is a sub-term of $\mathscr{S}_\mathrm{I}$, and removing the other threads:

**Definition 6.2.6 (configuration projection).** Configuration projection is defined as follows:

$$\Pi_\mathrm{C}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(C) = \{\Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(P,id,E) \mid (P,id,E) \in C,\ lab(P) \in \mathscr{L}_\mathrm{I}\}.$$

We now define the projection of a computation sequence. At each computation step, we obtain the set of the threads by projecting all the instances of syntactic components of $\mathscr{S}_\mathrm{I}$, and throwing away instances of syntactic components of the context. Unfortunately, the set of unsafe names, and the set of fresh markers, maynot be constructed without any knowledge of the previous computation steps, so we construct them incrementally: at each computation step, we insert spied names into the set of unsafe names, and remove the used markers from the set of fresh markers. We also ignore all computation steps only involving the context.

**Definition 6.2.7. (trace projection).** Computation sequence projection is then defined as follows: Let $\tau = C_0 \xrightarrow{\lambda_1}_\mathrm{e} \dots \xrightarrow{\lambda_n}_\mathrm{e} C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}_0^\mathrm{e}(\mathscr{S})$. We define the projection of $\tau$, $\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(\tau)$ as the non standard computation sequence:

$$(A_0,U_0,F_0) \overset{\Pi_\lambda(\lambda_{a_1})}{\looparrowright} \dots \overset{\Pi_\lambda(\lambda_{a_p})}{\looparrowright} (A_p,U_p,F_p)$$

of the open system $\mathscr{S}_\mathrm{I}$, where

- $a_1,\dots,a_p$ is the strictly ascending sequence of the elements of the set $\{i \in [\![1;n]\!] \mid \lambda_i \in \mathscr{L}^2 \setminus (\mathscr{L} \setminus \mathscr{L}_\mathrm{I})^2\}$;
- the initial configuration $(A_0,U_0,F_0)$ is the following triple:
$$(\Pi_\mathrm{C}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(C_0), en, \{t_n \mid n \in \mathbb{N}\});$$
- for $k \in [\![1;p]\!]$, the configuration $(A_k,U_k,F_k)$ is defined as follows:
  - $A_k = \Pi_\mathrm{C}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(C_{a_k})$,
  - $U_k = \begin{cases} U_{k-1} & \text{if } fst(\lambda_{a_k}) \in \mathscr{L}_\mathrm{I}, \\ U_{k-1} \cup \{\Pi_\mathscr{N}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(E(x_r)) \mid r \in [\![1;n]\!]\} & \text{otherwise,} \end{cases}$
  where, in the last case, $(x!^j[x_1,\dots,x_n]P,id,E)$ is the unique thread in $C_{a_{k-1}} \setminus C_{a_k}$ which matches this notation;
  - $F_k = \begin{cases} F_{k-1} \text{ if } \begin{cases} snd(\lambda_{a_k}) \in \mathscr{L}_\mathrm{I} \text{ or} \\ \xrightarrow{\lambda_{a_k}}_\mathrm{e} \text{ is not a resource fetching,} \end{cases} \\ F_{k-1} \setminus \{\Phi_\mathscr{M}(snd(\lambda_{a_k}),id) \mid (P,id,E) \in C_{a_k-1} \setminus C_{a_k}\} \text{ otherwise.} \end{cases}$

#### 6.2.3.2 Soundness

We first establish the soundness of the context independent semantics (i.e. the context independent semantics captures the abstraction of any trace for the computation of the open system in any context).

**Theorem 6.2.8. (Soundness)** *Let $\tau = C_0 \dots C_n$ be a non standard computation sequence of the following closed system:*

$$\mathscr{S} = (\nu\, c_1) \dots (\nu\, c_k)(\mathscr{S}_{\mathrm{I}}(c_{i_1}, \dots, c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1}, \dots, c_{j_l})),$$

*with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$. Then $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = (A_0, U_0, F_0) \dots (A_p, U_p, F_p)$ is a non standard computation sequence of the open system $\mathscr{S}_{\mathrm{I}}$ and $(A_0, U_0, F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S})$.*

Soundness is ensured by construction. The full proof of Thms. 6.2.8 is shown in appendix C.

#### 6.2.3.3 Completeness

Then, we establish the completeness of the context independent semantics (i.e. any trace in the context independent semantics is the abstraction of a trace for the computation of the open system in a given context).

**Theorem 6.2.9. (Completeness)** *Let $\tau'$ be the non standard computation sequence of an open system $\mathscr{S}_{\mathrm{I}}$, that we denote by:*

$$(C_0, U_0, F_0) \overset{(i_1, j_1)}{\looparrowright} \dots \overset{(i_n, j_n)}{\looparrowright} (C_n, U_n, F_n),$$

*where $(C_0, U_0, F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S}_{\mathrm{I}})$.*

*Then, there exists:*

- *a closed system $\mathscr{S}_* = (\nu\, \overline{c})(\mathscr{S}_{\mathrm{I}}(c_{i_1}, \dots, c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1}, \dots, c_{j_l}))$,*
- *two one-to-one functions $\Phi_{\mathscr{N}}$ and $\Phi_{\mathscr{M}}$,*
- *a non standard computation sequence $\tau$ of the system $\mathscr{S}_*$,*

*such that $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = \tau'$.*

Completeness relies on the existence of a most general context which may be used in simulating any context. It is given in Fig. 6.2. It uses a global channel, named unsafe via which unsafe names are sent an arbitrary amount of times. It is made of four kinds of resources. The resource **new** opens a new unsafe channel; the resource **repli** is used to replicate the information that a channel name is unsafe, so that a context may use each unsafe name an arbitrary number of time; the resource $\mathbf{spy}_k$ collects an unsafe channel $c$, and receive through this channel a message of arity $k$; the resource $\mathbf{spoil}_k$ collects an unsafe channel $c$, and $k$ unsafe names, and sends the $k$ names through the channel $c$. Resources $\mathbf{spy}_0$ and $\mathbf{spoil}_0$ simply enforce some synchronization with the system. In Fig. 6.2, $n$ denotes the greatest arity of the messages occurring in the part of the system we analyze. Thus, the closed system may be assumed to be of the following form:

$$\begin{array}{l}\mathscr{S}_* = (\nu\ \text{unsafe})(\nu\ x_1)\ldots(\nu\ x_p)\\ (\text{unsafe}![x_1] \mid \ldots \mid \text{unsafe}![x_p] \mid \mathscr{S}_I(x_{i_1},\ldots,x_{i_n}) \mid \mathscr{S}_c(\text{unsafe})).\end{array}$$

The non standard computation sequence is obtained by mimicking spied and spoiled computation steps in $\mathscr{S}_c$. The full proof of Thm. 6.2.9 is shown in appendix C.

$$\begin{array}{rl}\mathscr{S}_c = & (\nu\ \text{new})\\ & (\ \mathbf{new} \mid \mathbf{repli}\\ & \mid \mathbf{spy}_0 \mid \ldots \mid \mathbf{spy}_n\\ & \mid \mathbf{spoil}_0 \mid\ \ldots\ \mid \mathbf{spoil}_n\\ & \mid \text{new}![]\\ & )\end{array}$$

where

- $\mathbf{new} := *\text{new}?[]((\nu\ \mathit{channel})(\text{unsafe}![\mathit{channel}] \mid \text{new}![]))$
- $\mathbf{repli} := *\text{unsafe}?[x](\text{unsafe}![x] \mid \text{unsafe}![x])$
- $\mathbf{spy}_i := *\text{unsafe}?[c]c?[y_1,\ldots,y_i](\text{unsafe}![y_1] \mid\ \ldots\ \mid \text{unsafe}![y_i])$
- $\mathbf{spoil}_i := *\text{unsafe}?[c]\text{unsafe}?[x_1]\ldots\text{unsafe}?[x_i]c![x_1,\ldots,x_i]$

Figure 6.2: The most general context.

## 6.3 Generalization for the meta language

We want to extend the framework that we have proposed in the previous section to any model that is compiled in our meta language. This way, we abstract the context by the set of the values it may use and by a set of fresh markers for the thread that it spawns. Using this abstraction, we derive a sound abstraction of the context behavior. Unlike the $\pi$-calculus case, this approach is not complete in general. This context abstraction may be easily encoded at the level of the meta language, so that further analyses may be designed without distinguishing context independent and closed semantics.

### 6.3.1 Context approximation

The context is a set of threads, concurrently running with the analyzed system. We represent this context by the set of the values that may be used by the context. We call such values the *unsafe values*. We approximate the behavior of the context as if it was an intruder who was able to compute any partial interaction that uses these unsafe values. Furthermore, whenever a partial interaction is computed by the context, the context gets new unsafe values. When fetching a resource, the markers and the labels of the threads that fetch the resource are required. However, the history of this thread is not insightful, so we assume (by using an injective function from $\mathscr{L} \times \mathscr{M}$ into $\mathscr{M}$) that the label of each context thread is the same and that the marker is picked in a set of fresh markers. The same way, we are not interested in the history of the creation of the values that are declared by the context. That is why we assume that those values are all created by recursive instances of a same thread. We choose[1] some fresh distinct labels $cont_?$, $cont_!$, and $ext \in \mathscr{L}_\mathrm{p}$ some fresh labels. The values that are created by the context are seen as if they had been declared with the static label $ext$ by a recursive instance of a thread the marker of which is $t_n$, where $t_n$ is recursively defined as follows:

$$\begin{cases} t_0 & = & N((cont_?,\ cont_!), \varepsilon, \varepsilon) \\ t_{n+1} & = & N((cont_?,\ cont_!), \varepsilon, t_n). \end{cases}$$

Thus, the set of the values opened by the context may be chosen as the set $\{(ext, t_n) \mid n \in \mathbb{N}\}$, which we denote by $en$. We also assume that all spoiling messages are recursive instances of a single thread the first action of which is labeled with $ext$ (so that the context may create pointers to the thread that it computes).

The coherence of our semantics mainly relies on the fact that during a computation sequence, there maynot be two different instances of a single thread with the same marker. We guarantee this property by associating to each spoiling message a fresh marker in the set $\{t_n \mid n \in \mathbb{N}\}$.

[1]We have assumed $ext \in \mathscr{L}_\mathrm{p}$ so that the context may build pointers to the thread it creates, we make no syntactic distinction between the names and the pointers that are created by the context

#### 6.3.1.1 Context knowledge

A non standard configuration is now a triple $(C,U,F)$, where $C$ is a set of threads, $U$ is a set of values, and $F$ is a set of markers. The set $C$ contains the running threads. The set $U$ contains all the value $(a,id)$ that are known by the context. The set $F$ contains fresh markers which have not been used as markers for spoiling messages. At the beginning of the system, free variables[2] have to be chosen among the set of initial unsafe values. We make no assumption about the past of the system, so that distinct free variables may be bound to the same unsafe value. This is especially useful when analyzing an instance of a resource without any knowledge of the relations between the values that have been communicated to it.

#### 6.3.1.2 Primitives

**6.3.1.2.1 Thread synthesis** The context may build any thread $(t,id,E)$ with an available marker and an environment that associates some variables to unsafe values. Let $N$ be the maximum parameter number of a partial interaction. We may assume without any loss of generality that the domain of $E$ is the set $\{S_i \mid 1 \leq i \leq N\}$ of distinct variables in $\mathscr{V}$.

We may now define the set $\textsc{context\_thread}(U,F)$ of the threads that may be built by a context that satisfies the abstraction $(U,F)$ as follows:

**Definition 6.3.1.** Let $U \in \wp(\mathscr{L} \times \mathscr{M})$ be a set of values. Let $F \in \wp(\mathscr{M})$ be a set of fresh markers. The set $\textsc{context\_thread}(U,F)$ is given by:

$$\left\{ (0,id,E) \;\middle|\; \begin{array}{l} id \in F \\ Dom(E) = \{S_i \mid 1 \leq i \leq N\} \\ \forall i \in [\![1;N]\!],\ E(i) \in U \end{array} \right\}.$$

**6.3.1.2.2 Interaction synthesis** We suppose that the threads that are created by the context may compute any kind of partial interactions that deals with unsafe values. We also suppose that the context only computes threads at program point 0 with a fresh marker. Moreover, we assume without any loss of generality that the context ignores guards. We ignore the continuation launched by the context, since it is modeled when updating the context knowledge.

This yields the following definition:

**Definition 6.3.2.** Let $pi = (s,(parameter_i),(bound_i),constraints,continuation)$ be a partial interaction. Let $(m,n) = arity(s)$ be the arities of $s$. We say that the partial interaction $pi$ may be computed by a thread in $\textsc{context\_thread}(U,F)$

[2] A variable $v$ is free, if it is in the interface of an initial program point, but not in the static environment of this initial program points (i.e. if $v \in \bigcup\{I(p) \setminus Dom(E_s) \mid (p,E_s) \in \bigcup init_s\}$).

and we write $\text{CONTEXT\_PARTIAL\_INT}(pi)$ if and only if the following properties are satisfied:

1. $(paramater_i) \in \{S_i \mid 1 \leq i \leq N\}^m$;
2. $(bound_j) \in \mathscr{V}^n$;
3. $constraints = \emptyset$;
4. $continuation = \{\emptyset\}$.

**6.3.1.2.3 Knowledge updating** During an interaction, the context gets all the values that are associated to one of the variable of its threads. We now introduce a primitive that describes the set of the values that are got by the context when performing a partial interaction. This primitive takes a thread index $i$, a tuple of threads, the tuple of the sequences of each thread parameters, and the value passing formal description. It returns the set of the values that are spied by the context.

**Definition 6.3.3 (spied values).** Let $m,n \in \mathbb{N}$ be two integers. Let $i$ be an integer such that $1 \leq i \leq n$. Let $(t^k)_{1\leq k\leq n} = (p^k, id^k, E^k)_{1\leq k\leq n}$ be an $n$-tuple of threads. Let $(bd_l)_{1\leq l\leq m}$ be a sequence of variables ($bd_l$ is associated with the $l$-th variable that is bounded by the partial interaction in the $i$-th thread). Let $(param_l^k)_{k,l}$ be an $n$-tuple of parameter sequences ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread). Let *communications* be a partial map from $\mathscr{V}_f^Y$ into $\mathscr{V}_f^X \cup \mathscr{V}_f^I$ such that $\{Y_l^i \mid 1 \leq l \leq m\} \subseteq Dom(communications)$, we define the set $\text{SPIEDVAL}(i,(t^k),(bd_l),(param_l^k),communications)$ by:

$$\begin{cases} \emptyset & \text{if } p^i \neq 0, \\ \{\sigma(communications(Y_j^i)) \mid 1 \leq j \leq m\} & \text{otherwise,} \end{cases}$$

where $\sigma = \begin{cases} X_l^k \mapsto E^k(param_l^k) \\ I^k \mapsto (p^k, id^k). \end{cases}$

**6.3.1.2.4 Marker consumption** To ensure marker allocation freshness, the context may not perform several partial interactions of type *computation* with the same thread marker. In the case when a partial interaction is not of type *computation*, we do not remove any marker: this allows the context to use pointers to its thread and to use pointed threads without consuming them (in a migration step for instance).

**Definition 6.3.4 (marker consumption).** Let $t$ be a partial interaction name type in $\{replication; computation; migration\}$. Let $id$ be a marker in $\mathscr{M}$. We define the set $\textsc{consum}(t,id)$ of the markers that are consumed when a thread with marker $id$ computes a partial interaction of type $t$ by:

$$\textsc{consum}(t,id) \triangleq \begin{cases} \{id\} & \text{if } t = computation \\ \emptyset & \text{otherwise.} \end{cases}$$

**6.3.1.2.5 Broadcast communication** The substitution is applied to model broadcast communications just after having launched thread continuations. This substitution can be computed as in the case of the semantics for closed systems. Nevertheless, this substitution also changes the knowledge of the context. We define a primitive that takes a substitution and the context knowledge. This primitive returns the context knowledge after having computed broadcast communications.

**Definition 6.3.5 (broadcast substitution of the context knowledge).** Let $\tau \in (\mathscr{L} \times \mathscr{M}) \to (\mathscr{L} \times \mathscr{M})$ be a substitution. Let $U \in \wp(\mathscr{L} \times \mathscr{M})$ be a set of unsafe values. We define the set $subs\_context(\tau, U) \in \wp(\mathscr{L} \times \mathscr{M})$ of unsafe values by:

$$subs\_context(\tau,U) \triangleq \{\tau(u) \mid u \in U\}.$$

#### 6.3.1.3 Transition system

We use these primitives in order to describe both initial states and the semantics of computation steps according to a formal rule. We consider an open system. We denote by $\mathscr{V}_0$ the set of its free variables; the set $\mathscr{V}_0$ is defined as $\bigcup\{I(p) \setminus Dom(E_s) \mid (p,E_s) \in \bigcup init_s\}$. Initial states are obtained by launching a continuation in $init_s$ with an empty marker and an environment that maps each initial free variable $v \in \mathscr{V}_0$ to an unsafe value $u \in en$. Thus the set $\mathscr{C}_0^o$ of the initial state is defined as:

$$\mathscr{C}_0^o = \left\{ (launch(continuation, \varepsilon, E_0), en, \{t_n \mid n \in \mathbb{N}\}) \;\middle|\; \begin{array}{l} continuation \in init_s, \\ E_0 \in \mathscr{V}_0 \to en \end{array} \right\}.$$

Computation steps are described by a reduction relation in Fig. 6.3. We recall the different steps of this computation, as follows:

- *interaction enabling*:
  - first, we find some threads that exhibit the right partial interactions (these threads either belong to the system, or are built by the context);
  - then, we check that their interface is compatible with the synchronization constraints in the formal rule;

- *interaction computation*:
  - we remove the threads that do not compute a replication;
  - we choose a syntactic continuation for each thread;
  - we compute dynamic data for each of these continuations:
    - we compute the marker;
    - we take into account name passing;
    - we create fresh variables and associate them with the correct values;
    - we restrict the environment to the interface of the thread that is launched;
  - we update the set of unsafe names;
  - we update the set of available markers;
  - we apply broadcast substitution to both the system threads and the unsafe names in order to model potential re-addressing.

Each computation step is labeled with some information we need to update the state of the system. More precisely, a computation step label is of the form $(\mathscr{R},((t^1,pi^1,Ct^1),\ldots,(t^n,pi^n,Ct^n)),\tau(i))$ where $\mathscr{R}$ is a reduction rule which expects $n$ interacting threads; where, for any $k \in [\![1;n]\!]$, the $k$-th thread that is involved in the interaction is the thread $t^k$, this thread computes the partial interaction $pi^k$ before launching the continuation $Ct^k$; and where $\tau$ is the broadcast substitution that is applied to any thread at the end of the computation step.

### 6.3.2 Soundness

We propose to establish a relation between the non standard semantics of closed and open systems. We consider an open system $\mathscr{S}_{\mathrm{I}}$. We recall that the set $\mathscr{V}_0 = \bigcup\{I(p)\setminus Dom(E_s) \mid (p,E_s) \in \bigcup init_s\}$ is the set of the free variables of the mobile system $\mathscr{S}_{\mathrm{I}}$. We denote by $\mathscr{L}_{\mathrm{p}}^{\mathrm{I}}$ the set of the labels of the program points of the system $\mathscr{S}_{\mathrm{I}}$. Let $\mathscr{S}$ be a mobile system. We denote by $(I, init_s, interaction)$ the syntax of the system $\mathscr{S}$ and by $(I^{\mathrm{I}}, init_s^{\mathrm{I}}, interaction^{\mathrm{I}})$ the syntax of the system $\mathscr{S}_{\mathrm{I}}$. We say that the system $\mathscr{S}$ encloses the open system $\mathscr{S}_{\mathrm{I}}$ if and only if the two following conditions are satisfied:

- The syntax of $\mathscr{S}$ satisfies the sufficient properties in Def. 4.5.11.
- The syntax of $\mathscr{S}$ coincides with the syntax of $\mathscr{S}_{\mathrm{I}}$. This means that:
  1. $I_{|\mathscr{L}_{\mathrm{p}}^{\mathrm{I}}} = I^{\mathrm{I}}$,

Let $(C,U,F)$ be a configuration.
Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule.
Let $\mathscr{K}_c$ be a subset of $[\![1;n]\!]$. We denote $\mathscr{K}_s = [\![1;n]\!] \setminus \mathscr{K}_c$.
If there exists $(t^k)_{1\leq k\leq n} = (p^k, id^k, E^k)_{1\leq k\leq n} \in C^n$ a sequence of distinct threads, and $(pi^k)_{1\leq k\leq n} = (s^k, (parameter^k_l)_l, (bd^k_l)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$ an $n$-tuple of partial interactions, such that:

1. - $\forall k \in \mathscr{K}_s$, $exhibit(t^k, pi^k)$;
   - $\forall k,k' \in \mathscr{K}_c$, $\begin{cases} t^k \in \text{CONTEXT\_THREAD}(U,F), \\ \text{CONTEXT\_PARTIAL\_INT}(pi^k), \\ id_k = id_{k'} \implies k = k'; \end{cases}$
2. $\forall k \in [\![1;n]\!]$, $components(k) = s^k$;
3. $sync((t^1,\ldots,t^n), (parameter^k_l)_{k,l}, compatibility)$ is satisfied.

Then:

$$C \overset{(\mathscr{R},(\alpha^1,\ldots,\alpha^n),\tau)}{\hookrightarrow} (subs(\tau, C \setminus removed\_threads \cup new\_threads), subs\_context(\tau,U'), F')$$

where:

1. $\tau \in subs\_choice((t^k)_k, (parameter^k_l)_{k,l}, broadcast)$;
2. $removed\_threads = remove((t^k, type(s^k))_{k\in\mathscr{K}_s})$;
3. $new\_threads = \bigcup\{launch(Ct^k, \overline{id}^k, \overline{E}^k) \mid k \in \mathscr{K}_s\}$,
   with $\forall k \in \mathscr{K}_s$:
   - $Ct_k \in continuation^k$;
   - $\overline{id}^k = marker(type(s_k), \left(p^{k'}, id^{k'}, E^{k'}\right)_{1\leq k'\leq n}, k)$;
   - $\overline{E}^k = vpassing(k, (t^{k'})_{k'}, (bd^k_l)_l, (parameter^{k'}_l)_{k',l}, communications)$.
4. $U' = U \cup \overline{U}$;
5. $\overline{U} = \bigcup\{\text{SPIEDVAL}(k, (t^{k'})_{k'}, (bd^k_l)_l, (parameter^{k'}_l)_{k',l}, communication) \mid k \in \mathscr{K}_c\}$;
6. $F' = F \setminus (\bigcup\{\text{CONSUM}(type(s^k), id^k) \mid k \in \mathscr{K}_c\})$;
7. - $\forall k \in \mathscr{K}_s$, $\alpha_k = (t^k, pi^k, Ct_k)$;
   - $\forall k \in \mathscr{K}_c$, $\alpha_k = (t^k, pi^k, \emptyset)$.

Figure 6.3: Generic transition rule.

2. for any continuation thread $(p,E_s) \in \bigcup(\bigcup(init_s))$, such that $p \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}$, there exists a continutation thead $(p',E_s') \in \bigcup\left(\bigcup init_s^{\mathrm{I}}\right)$ such that $p = p'$ and $E_s'(x) = E_s(x)$ for any $x \in Dom(E_s')$.

- Computations are compatible with the thread partition: this means that for any program point pair $(p,q)$, if there exist a partial interaction $pi = (s,(parameter_i),(bound_i), constraints, continuation)$ in $interaction(p)$ and a static environment $E_s$ such that $(q,E_s) \in \bigcup continuation$, then we have $p \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}$ if and only if $q \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}$.

We want to construct a projection function $\Pi_\tau$, such that any non standard computation sequence $\tau$ of a closed system that encloses the system $\mathscr{S}_{\mathrm{I}}$ is mapped to a non standard computation sequence $\Pi_\tau(\tau)$ of the open system $\mathscr{S}_{\mathrm{I}}$;

#### 6.3.2.1 Trace abstraction

Let $\mathscr{L}_{\mathrm{p}}^{\mathrm{I}} \subseteq \mathscr{L}$ be the subset of labels of the program points of the system $\mathscr{S}_{\mathrm{I}}$ and $\mathscr{N}_{\mathrm{I}} \subseteq \mathscr{L}$ be the subset of the name labels that are used in the static environments in continuations threads in the system $\mathscr{S}_{\mathrm{I}}$ in $syntax_{\mathrm{I}}$. We introduce two one-to-one functions in order to interpret names and threads created by the context of $\mathscr{S}_{\mathrm{I}}$: let $\Phi_{\mathscr{M}}$ be a one-to-one map from the set $(\mathscr{L} \times \mathscr{M})$ into the set $\{t_n \mid n \in \mathbb{N}\}$, and $\Phi_{\mathscr{N}}$ be a one-to-one function from the set $(\mathscr{N} \times \mathscr{M})$ into the set $en$. We now define the projection $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})$ which transforms each computation sequence of any closed mobile system that encloses the system $\mathscr{S}_{\mathrm{I}}$ into a computation sequence for the open system $\mathscr{S}_{\mathrm{I}}$. We assume without any loss of generality that $\mathscr{V}_0 = \{c_{i_k} \mid k \in [\![1;n]\!]\}$, and that no variable occurs twice as an argument of a variable binder.

We first project each syntactic component label, by replacing each label which does not occur in $\mathscr{S}_{\mathrm{I}}$ by the unique label of the context, that is to say the 0 label.

**Definition 6.3.6 (program point label projection).** The projection $\Pi_{\mathrm{l}}(l)$ of a syntactic component label $l \in \mathscr{L}$ is defined as follows:

$$\Pi_{\mathrm{l}}(l) = \begin{cases} l & \text{if } l \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}} \\ 0 & \text{otherwise.} \end{cases}$$

Next, we project the instance markers of syntactic components of $\mathscr{S}_{\mathrm{I}}$. Each marker that denotes the history of a context thread is replaced by its image by $\Phi_{\mathscr{M}}$. Each marker that denotes the history of a thread of the system is replaced by applying recursively marker projection to its sub-markers.

**Definition 6.3.7 (marker projection).** Marker projection is defined as follows:

$$\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(N((l_1,\ldots,l_n),t_1,\ldots,t_n)) = N((\Pi_{\mathrm{l}}(l_1),\ldots,\Pi_{\mathrm{l}}(l_n)),h(l_1,t_1),\ldots,h(l_n,t_n)),$$

where $h(l,id) = \begin{cases} \Phi_{\mathscr{M}}(l,id) & \text{if } l_i \notin \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}, \\ \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(t_i) & \text{otherwise.} \end{cases}$

We now project values. For any value the label of which belongs to the system, we just project the marker. For any value that has been declared by the context, we replace the label with the label *ext* and we compute the coherent marker according to $\Phi_{\mathscr{N}}$.

**Definition 6.3.8 (value projection).** Value projection is defined as follows:

$$\Pi_{\mathscr{L}\times\mathscr{M}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(x,id_x) = \begin{cases} (x,\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_x)) & \text{if } x \in \mathscr{N}_{\mathrm{I}} \\ \Phi_{\mathscr{N}}(x,id_x) & \text{otherwise.} \end{cases}$$

We can easily project an instance of a syntactic component of $\mathscr{S}_{\mathrm{I}}$ by projecting its marker and each value that occurs in its environment.

**Definition 6.3.9 (thread projection).** Thread projection is defined as follows:

$$\Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(P,id,E) = (P,\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id),[x \mapsto \Pi_{\mathscr{L}\times\mathscr{M}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(x))]).$$

Then, we can project a configuration by projecting all the threads at program point in $\mathscr{L}_{\mathrm{p}}^{\mathrm{I}}$ and by removing the other threads:

**Definition 6.3.10 (configuration projection).** Configuration projection is defined as follows:

$$\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C) = \{\Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(p,id,E) \mid (p,id,E) \in C,\ P \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}\}.$$

Now, we project broadcast substitution by using the value projection which is injective:

**Definition 6.3.11 (substitution projection).** Let $\tau = [u_i \mapsto v_i, i \in I]$ a broadcast substitution, where $I$ is a finite set, $(u_i)$ is a family of pair-wise distinct values, and $(v_i)$ is a family of values. The projection $\Pi_{\mathrm{SUBS}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(\tau)$ is defined as:

$$[\Pi_{\mathscr{L}\times\mathscr{M}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(u_i) \mapsto \Pi_{\mathscr{L}\times\mathscr{M}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(v_i)].$$

Then, we can define transition label projection by applying both label projection and substitution projection component-wise and by removing the continuation of the threads of the context:

**Definition 6.3.12 (transition label projection).** The projection $\Pi_\lambda(\mathscr{R},(p^i,pi^i,Ct^i)_i,\tau)$ of transition label $(\mathscr{R},((p^i,pi^i,Ct^i)_i),\tau)$ is defined as follows:

$$\Pi_\lambda(\mathscr{R},(p^i,pi^i,Ct^i)_i,\tau) = (\mathscr{R},(\Pi_{\mathrm{l}}(p^i),pi^i,h(p^i,Ct^i))_i,\Pi_{\mathrm{SUBS}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(\tau)),$$

where $h(p,Ct) = \begin{cases} Ct^i & \text{if } p \in \mathscr{L}_{\mathrm{p}}^{\mathrm{I}}, \\ \emptyset & \text{otherwise.} \end{cases}$

We can now define the projection of a computation sequence. At each computation step, we obtain the set of threads by projecting all instances of syntactic components of $\mathscr{S}_{\mathrm{I}}$ and by throwing away instances of syntactic components of the context. Unfortunately, the set of values and the set of fresh markers cannot be constructed without any knowledge of the previous computation steps, so we construct them incrementally: at each computation step, we insert spied values into the set of unsafe values and remove the used markers from the set of fresh markers. We also apply the broadcast subsitution to each unsafe names.

**Definition 6.3.13. (trace projection).** Computation sequence projection is then defined as follows: Let $\tau = C_0 \xrightarrow{\lambda_1} \ldots \xrightarrow{\lambda_n} C_n$ be a non standard computation sequence in the closed semantics of a mobile system that encloses the open system $\mathscr{S}_{\mathrm{I}}$, with $C_0 \in \mathscr{C}_0$. We define the projection of $\tau$, $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau)$ as the non standard computation sequence:

$$(A_0, U_0, F_0) \overset{\Pi_\lambda(\lambda_{a_1})}{\looparrowright} \ldots \overset{\Pi_\lambda(\lambda_{a_n})}{\looparrowright} (A_n, U_n, F_n)$$

of the open system $\mathscr{S}_{\mathrm{I}}$, where

- the initial configuration $(A_0, U_0, F_0)$ is the following triple:

$$(\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_0), en, \{t_n \mid n \in \mathbb{N}\});$$

- for $k \in [\![1; n]\!]$,
  we denote $\lambda_k = (\mathscr{R}, (t^i, pi^i, Ct^i), \tau^i)$,
  we denote $(t^i) = (p^i, id^i, E^i)$,
  we denote $(pi^i) = (s^i, (param^i_j)_j, (bd^i_j)_j, constraints^i, continuation^i)$,
  we denote $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$;
  then, the configuration $(A_k, U_k, F_k)$ is defined as follows:

  – $A_k = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_k)$,

  – $U_k = subs(U_{k-1} \cup \textsc{spy})$
    where:

    * $\textsc{spy} = \{\Pi_{\mathscr{L} \times \mathscr{M}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(\sigma(v\text{-}passing(Y^i_l))) \mid i \notin \mathscr{L}^{\mathrm{I}}_{\mathrm{p}}\}$,
      where $\sigma(X^i_l) = E_i(param^i_l)$ and $\sigma(I^i) = (p^i, id^i)$,
    * $subs = \Pi_{\textsc{subs}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(\tau^k)$,

  – $F_k = F_{k-1} \setminus \{\Phi_{\mathscr{M}}(p^i, id^i) \mid p^i \notin \mathscr{L}^{\mathrm{I}}_{\mathrm{p}},\ type(pi^i) = computation\}$.

#### 6.3.2.2 Soundness

We establish the soundness of the context independent semantics (i.e. the context independent semantics captures the abstraction of any trace for the computation of the open system in any context).

**Theorem 6.3.14. (Soundness)** *Let $\tau = C_0 \dots C_n$ be a non standard computation sequence of a closed system that encloses the open system $\mathscr{S}_{\mathrm{I}}$, with $C_0 \in \mathscr{C}_0$. Then $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = (A_0, U_0, F_0) \dots (A_n, U_n, F_n)$ is a non standard computation sequence of the open system $\mathscr{S}_{\mathrm{I}}$ and $(A_0, U_0, F_0) \in \mathscr{C}_0^{o}$.*

Soundness is ensured by construction. The full proof of Thm. 6.3.14 is shown in appendix C.

### 6.3.3 Implementation at the meta language level

We now encode our context abstraction at the meta language level.

#### 6.3.3.1 Unsafe values

We model the set of unsafe values by using some factious program points and some extra partial interactions. More precisely, we introduce a fresh program point label $p^{pick}$. The program point $p^{pick}$ is associated with the interface $\{X\}$. A thread at program point $p^{pick}$ and with the environment $[X \mapsto (x, id)]$ means that the context may use the unsafe value $(x, id)$ to build its own threads. A thread at program point $p^{pick}$ may compute a partial interaction names *pick* that allows for picking the unsafe value. More precisely, the arity of the name *pick* is $(1, 0)$ and its type is *computation.* A thread at program point $p^{pick}$ may compute the partial interaction $(pick, [X], [], \emptyset, \{\emptyset\})$.

Each unsafe values may be used several times, and even several times simultaneously. So, we have to spawn an unbounded number of thread instances at program point $p^{pick}$ for each unsafe value. We introduce a sequence of program point labels $(p_n^{unsafe})_{n \in \mathbb{N}}$. Each program point $p_n^{unsafe}$ is associated with the interface $\{X_n\}$. A thread at the program point $p_n^{unsafe}$ and with the environment $[X_n \mapsto (x, id)]$ is a resource that may spawn an unbounded number of thread instances at program point $p^{pick}$ with the environment $[X \mapsto (x, id)]$. The meaning of the index $n$ will be explained later. (It means that this value has become unsafe, because it has be bound to the $n$-th bound variable in a partial interaction). A thread at program point $p_n^{unsafe}$ may compute a partial interaction named *duplicate* that allows for spawning a thread at program point $p^{pick}$ where the variable $X$ is associated with the value of the variable $X_n$. More precisely, the arity of the name *duplicate* is $(1,1)$ and its type is *replication.* A thread at program point $p_n^{unsafe}$ may compute the partial interaction $(duplicate,[X_n],[X],\emptyset,\{\{(p^{pick},\emptyset)\}\})$.

Then, we introduce the program point $p^{fetch}$ for the threads that fetch the resources at program point $p_n^{unsafe}$. The program point $p^{fetch}$ is associated with the interface $\emptyset$. Threads at program point $p^{fetch}$ may compute a partial interaction named $\overline{duplicate}$. The arity of the name $\overline{duplicate}$ is $(0,0)$ and its type is *computation.* A thread at program point $p^{fetch}$ may compute the partial interaction $(\overline{duplicate},[],[],\emptyset,\{\emptyset\})$. We give in Fig. 6.4 the formal rule that allows for the replication of unsafe values. The communication $[Y_1^1 \mapsto X_1^1]$ describes the fact that the value of the variable $X_n$ is passed to the variable $X$ at program point $p^{pick}$.

$$rec = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto duplicate \\ 2 \mapsto \overline{duplicate} \end{cases}$
2. $synchronization = \emptyset$;
3. $communication = [Y_1^1 \leftarrow X_1^1]$;
4. $global = \emptyset$.

Figure 6.4: Formal rule for duplicating unsafe values.

In order to spawn an unbounded number of threads at program point $p^{fetch}$, we introduce two program points $p_{resource}^{fetch}$ and $p_{fetch}^{fetch}$. Each time a thread at program point $p_{resource}^{fetch}$ interacts with a thread at program point $p_{fetch}^{fetch}$, two threads are launched: the first one at program point $p_{fetch}^{fetch}$ and the second one at program point $p^{fetch}$. Thus, we introduce two names *fetch* and $\overline{fetch}$ of partial interactions. the arity of these names are $(0,0)$, the name *fetch* is of type *replication*, and the name $\overline{fetch}$ is of type *computation.* A thread at program point $p_{resource}^{fetch}$ has an empty interface and is associated with the set of partial interactions $\{(fetch,[],[],\emptyset,\{\{(p^{fetch},\emptyset);(p_{fetch}^{fetch},\emptyset)\}\})\}$. A thread at program point $p_{fetch}^{fetch}$ has an empty interface and is associated with the set of partial interactions $\{(\overline{fetch},[],[],\emptyset,\{\emptyset\})\}$. We give in Fig. 6.5 the formal rule that allows for the replication of the resource at program point $p_{resource}^{fetch}$.

We now describe another resource. This resource aims at copying name tuples that are learnt from the spied system. We introduce a program point label $p^{wait}$

$$rec = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 \mapsto fetch \\ 2 \mapsto \overline{fetch} \end{cases}$

2. $synchronization = \emptyset$;

3. $communication = []$;

4. $global = \emptyset$.

Figure 6.5: Formal rule for unfolding the resource.

and a family of partial interaction names $(learn_n)_{n\in\mathbb{N}}$. For any integer $n \in \mathbb{N}$, the partial interaction name $learn_n$ is of type *replication* and have the arity $(0,n)$. Threads at program point $p^{wait}$ has an empty interface and is associated with the following set of partial interactions:

$$\{(learn_n, [], [X_1, \dots, X_n], \emptyset, \{(p_i^{unsafe}, \emptyset) \mid 1 \leq j \leq n\}) \mid n \in \mathbb{N}\}$$

When an $n$-tuple is received, this resource launches threads at program points $p_i^{unsafe}$ with an environment that maps the variable $X_i$ to the $i$-th component of the tuple (Cf. Fig. 6.8 on page 147).

#### 6.3.3.2 Context instance

The context may launch a context instance for any fresh marker in $F$. We introduce some factious program points, some fresh partial interaction names, and some new formal rules. We introduce the program point $p^{available}$. For each integer $n \in \mathbb{N}$, we introduce the program points $p_n^{consume}$ and $p_n^{recycle}$. We also introduce the partial interaction names *spy* and *release*. Moreover, for each partial interaction name *pi* of the closed semantics, we introduce both a new program point $p^{pi}$ and a fresh name of partial interaction $pi'$.

An informal description of the behavior of a context thread instance is given in Fig. 6.6. In this figure, we have assumed that there are only two partial interaction kinds in the closed semantics. Partial interactions named $pi_1$ are of type *computation* or *replication*, and partial interactions named $pi_2$ is of type *migration*.

**6.3.3.2.1 Computing a partial interaction** More precisely, a thread at program point $p^{available}$ and with the marker *id* denotes the fact that *id* is available in $F$, its interface is empty. A thread at program point $p^{pi}$ denotes the fact

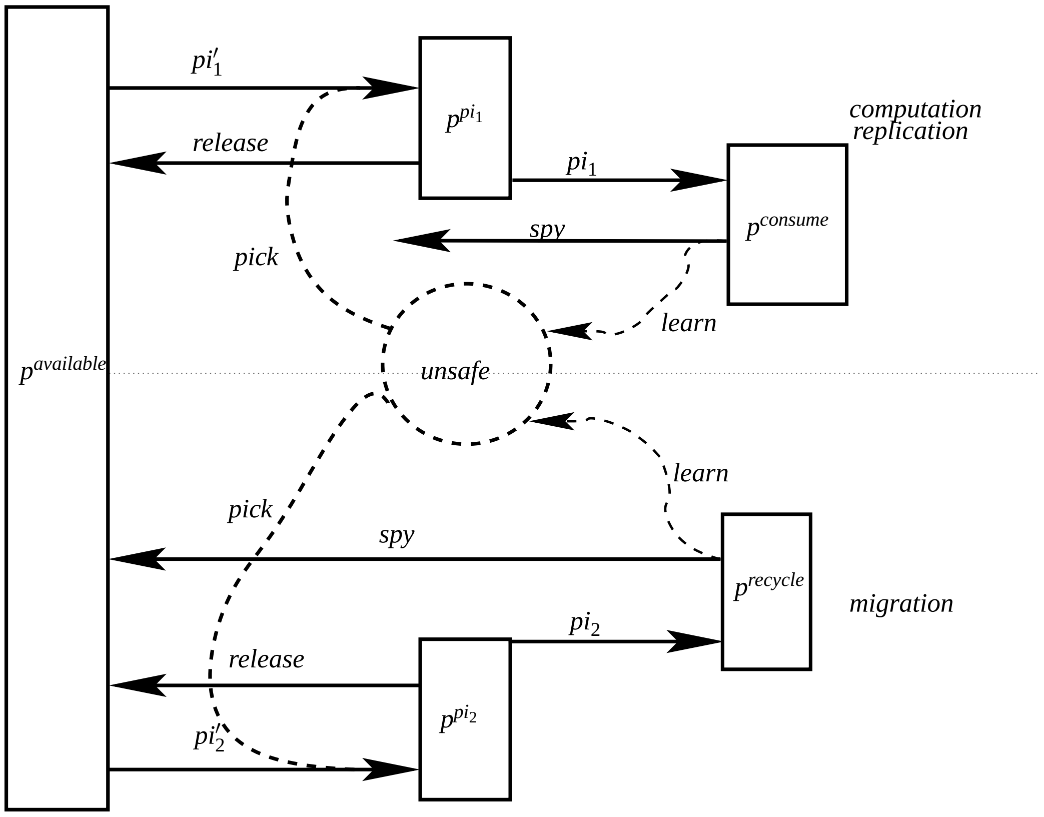


Figure 6.6: Context thread instance behavior.

that a thread is using a thread marker and that this thread may compute the partial interaction $pi$. The interface of this thread is the set $\{c_i \mid 1 \leq i \leq m\}$, where $Ari(pi) = (m,n)$. Each variable $c_i$ denotes a parameter of the partial interaction $pi$. A thread at program point $p^{available}$ may reach the program point $p^{pi}$ by computating the partial interaction $pi'$. Let $pi$ be a partial interaction name. Let $(m,n) = Ari(pi)$. We define $Ari(pi') = (0,m)$ and $type(pi') = type(pi)$. We define the formal rule $\mathscr{R}_{pi}$ in Fig. 6.7. The formal rule $\mathscr{R}_{pi}$ that gives the capability to build threads to the context.

$$\mathscr{R}_{pi} = (m+1, component, synchronization, communication, global)$$

where

1. $(m,n) = type(pi)$;
2. $component = \begin{cases} i & \mapsto pick \text{ if } i \leq m \\ m+1 & \mapsto pi'; \end{cases}$
3. $synchronization = \emptyset$;
4. $communication = [Y_j^{m+1} \leftarrow X_1^j, 1 \leq j \leq m]$;
5. $global = \emptyset$.

Figure 6.7: Context thread building.

A thread at program point $p^{available}$ is associated with the set $\{(pi', [], [c_1, \ldots, c_m], \{(p^{pi}, \emptyset)\} \mid pi \in \mathscr{A}, (m,n) = Ari(pi)\})$ of partial interactions.

Once it has reached a program point $p^{pi}$ (we denote $(m,n) = Ari(pi)$), the instance may compute a partial interaction $pi$ or release the thread marker. Releasing the thread marker is mandatory in the case when the partial interaction $pi$ is of type *replication*, since the thread marker may be re-used after having replicated the resource. When the thread computes the partial interaction $pi$, it has to declare new unsafe values. That is the purpose of threads at program point $p^{consume_n}$ and $p^{recycle_n}$: a thread at program point $p^{consume_n}$ declares $n$ unsafe values and launches no continuation (the instance marker is consummed), whereas a thread at program point $p^{recycle_n}$ declares $n$ unsafe values and releases the instance marker. So there are several cases according to the type of the partial interaction. In the case when $type(pi) \in \{computation; replication\}$, the program point $p^{pi}$ is associated with the following set of partial interactions:

$$\left\{ \begin{array}{l} (pi, [c_1, \ldots, c_m], [X_1, \ldots, X_n], \emptyset, \{\{(p^{consume_n}, \emptyset)\}\}); \\ (release, [], [], \emptyset, \{\{(p^{available}, \emptyset)\}\}) \end{array} \right\}.$$

Otherwise, the program point $p^{pi}$ is associated with the following set of partial interactions:

$$\left\{ \begin{array}{l} \{(pi,[c_1,\ldots,c_m],[X_1,\ldots,X_n],\emptyset,\{\{(p^{recycle_n},\emptyset)\}\})\}; \\ (release,[],[],\emptyset,\{\{(p^{available},\emptyset)\}\}) \end{array} \right\}$$

A thread at program point $p^{consume_n}$ or $p^{recycle_n}$ has the interface $\{X_i \mid 1 \leq i \leq n\}$.

**6.3.3.2.2 Learning new unsafe names** We now describe how the context enlarges the set of unsafe values. The context learn new unsafe names when a thread at program point $p^{wait}$ interact with a thread at program point $p^{consume_n}$ or $p^{recycle_n}$. We introduce a family of partial interaction names $(spy_n)$. We define $type(spy_n) = migration$ and $Ari(spy_n) = (n,0)$. A thread at program point $p^{consume_n}$ is associated with the set $\{(spy_n,[X_1,\ldots,X_n],[],\emptyset,\{\emptyset\})\}$ of partial interactions and a thread at program point $p^{release_n}$ is associated with the set $\{(spy_n,[X_1,\ldots,X_n],[],\emptyset,\{\{(p^{available},\emptyset)\}\})\}$ of partial interactions. We notice that a thread at program point $p^{release_n}$ does not consume the thread marker, since it launches a thread at program point $p^{available}$ whenever it computes a partial interaction. The formal rule $\mathscr{R}_{spy_n}$ describes the interaction between a thread at program point $p^{wait}$ interact with a thread at program point $p^{consume_n}$ or $p^{recycle_n}$. This formal rule is defined in Fig. 6.8.

$$\mathscr{R}_{spy_n} = (2, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 & \mapsto learn_n; \\ 2 & \mapsto spy_n; \end{cases}$
2. $synchronization = \emptyset$;
3. $communication = [Y_i^1 \leftarrow X_i^2,\ 1 \leq i \leq n]$;
4. $global = \emptyset$.

Figure 6.8: Value learning.

**6.3.3.2.3 Releasing thread markers** Only partial interaction of type *computation* consumes their instance marker. Instance marker of migrating threads may be release after the migration by applying a partial interaction named $spy_n$ (that launches a thread at program point $p^{available}$ whenever the computed partial interaction is of type *migration*). Unfortunately, we cannot apply this scheme for partial interaction of type *replication*. That is why we allows a thread at program point $p^{pi}$ to release its marker. That is the purpose of the partial interaction name *release*. When the context instance compute a partial interaction *pi* of type *replication*, the thread at program point $p^{pi}$ is not consumed, and the marker may be released by applying a partial interaction named *release*. The formal rule $\mathscr{R}_{release}$, which is given in Fig. 6.9 allows the release of a thread marker.

$$\mathscr{R}_{release} = (1, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} 1 & \mapsto release; \end{cases}$
2. $synchronization = \emptyset$;
3. $communication = \emptyset$;
4. $global = \emptyset$.

Figure 6.9: Context thread releasing.

#### 6.3.3.3 Initialization

The context may use an unbounded number of fresh markers and build an unbounded number of unsafe values. For that purpose, we introduce two resources that build context fresh markers and fresh unsafe values.

First, we introduce four program points: $p^{value}_{resource}$, $p^{marker}_{resource}$, $p^{value}_{fetch}$, and $p^{marker}_{fetch}$. Threads at any of these four program points have an empty environment. Then, we introduce four partial interaction names *build-value*, $\overline{\textit{build-value}}$, *build-marker* and $\overline{\textit{build-marker}}$. We set:

$$Ari : \begin{cases} \textit{build-value} & \mapsto (0,1), \\ \overline{\textit{build-value}} & \mapsto (0,0), \\ \textit{build-marker} & \mapsto (0,0), \\ \overline{\textit{build-marker}} & \mapsto (0,0); \end{cases} \qquad type : \begin{cases} \textit{build-value} & \mapsto replication, \\ \overline{\textit{build-value}} & \mapsto computation, \\ \textit{build-marker} & \mapsto replication, \\ \overline{\textit{build-marker}} & \mapsto computation. \end{cases}$$

Thus, these partial interactions have no parameter, they bind no variable, except the partial interaction *build-value* that associates a variable with the newly created value.

Threads at program point $p^{value}_{resource}$ may compute the following partial interaction:

$$(\textit{build-value}+, [], [X_1], \emptyset, \{\{(p^{unsafe}_1, \emptyset); (p^{value}_{fetch}, \emptyset)\}\});$$

threads at program point $p^{marker}_{resource}$ may compute the following partial interaction:

$$(\mathit{build\text{-}marker}+, [], [], \emptyset, \{\{(p^{available}, \emptyset); (p^{marker}_{fetch}, \emptyset)\}\});$$

threads at program point $p^{value}_{fetch}$ may compute the partial interaction $(\overline{\mathit{build\text{-}value}}, [], [], \emptyset, \{\emptyset\})$; and threads at program point $p^{marker}_{fetch}$ may compute the partial interaction $(\overline{\mathit{build\text{-}marker}}, [], [], \emptyset, \{\emptyset\})$.

We now define in Figs. 6.10 and 6.11 two formal rules $\mathscr{R}_{\mathit{build\text{-}value}}$ and $\mathscr{R}_{\mathit{build\text{-}marker}}$ to respectively build fresh values and fresh markers. We notice

$$\mathscr{R}_{\mathit{build\text{-}value}} = (2, \mathit{component}, \mathit{synchronization}, \mathit{communication}, \mathit{global})$$

where

1. $\mathit{component} = \begin{cases} 1 & \mapsto \mathit{build\text{-}value} \\ 2 & \mapsto \overline{\mathit{build\text{-}value}}; \end{cases}$
2. $\mathit{synchronization} = \emptyset$;
3. $\mathit{communication} = [Y^1_1 \leftarrow I^2]$;
4. $\mathit{global} = \emptyset$.

Figure 6.10: Fresh value declaration.

$$\mathscr{R}_{\mathit{build\text{-}marker}} = (2, \mathit{component}, \mathit{synchronization}, \mathit{communication}, \mathit{global})$$

where

1. $\mathit{component} = \begin{cases} 1 & \mapsto \mathit{build\text{-}marker} \\ 2 & \mapsto \overline{\mathit{build\text{-}marker}}; \end{cases}$
2. $\mathit{synchronization} = \emptyset$;
3. $\mathit{communication} = \emptyset$;
4. $\mathit{global} = \emptyset$.

Figure 6.11: Fresh marker declaration.

that when applying the rule $\mathscr{R}_{\mathit{build\text{-}value}}$ the fresh value is defined as the identity of the thread that fetch the resource.

#### 6.3.3.4 Initial state

We now describe the set of initial states of the system. We consider an open system $\mathscr{S}$. We suppose that the set of the free variable of $\mathscr{S}$ (i.e. $\bigcup\{I(p) \setminus Dom(E_s) \mid (p, E_s) \in \bigcup init_s\}$) is given by the set $\{v_i \mid 1 \leq i \leq m\}$ (where $m \in \mathbb{N}$). An initial state is given first by choosing the value of each free variable, and then by choosing the initial threads.

We introduce a program point $p^{init}$, a partial interaction name *init* and a formal rule $\mathscr{R}_{init}$ to model the initialization of the system. Threads at program point $p^{init}$ have an empty interface. Each partial interaction *init* have the arity $(0, m)$ and the type *computation*. Threads at program point $p^{init}$ may compute the partial interaction $(init, [], [v_1, \ldots, v_m], \emptyset, \mathscr{C}_0)$. This partial interaction picks $m$ unsafe values and launches an initial state of the open system.

$$\mathscr{R}_{init} = (m+1, component, synchronization, communication, global)$$

where

1. $component = \begin{cases} i & \mapsto pick \text{ if } i \leq m \\ m+1 & \mapsto pi'; \end{cases}$

2. $synchronization = \emptyset$;

3. $communication = [Y_j^{m+1} \leftarrow X_1^j, 1 \leq j \leq m]$;

4. $global = \emptyset$.

Figure 6.12: Context initialization.

The initial states of the system that simulates the computation of the open system $\mathscr{S}$ in any context is given as follows:

$$\left\{ \left\{ \begin{array}{l} (p^{init}, \emptyset); (p^{fetch}_{resource}, \emptyset); (p^{wait}, \emptyset); \\ (p^{fetch}_{fetch}, \emptyset); (p^{value}_{resource}, \emptyset); (p^{value}_{fetch}, \emptyset); (p^{marker}_{resource}, \emptyset); (p^{marker}_{fetch}, \emptyset) \end{array} \right\} \right\}.$$

#### 6.3.3.5 Marker unambiguity

Our context encoding satisfies marker unambiguity sufficient conditions (Cf. 4.5.11 on page 4.5.11).

### 6.3.4 Incompleteness

Our context abstraction is complete in the $\pi$-calculus (see Thm. 6.2.9). Unfortunately, this is not the case in general, because it gives too much power to the context. We isolate two causes of potential incompleteness and give concrete examples.

#### 6.3.4.1 The case of the spi-calculus

The first cause of incompleteness is due to migrating threads. When a thread migrates, its environment may change (or not), but its syntactic skeleton is fixed once for all. For instance, in the case of the spi-calculus, when a term $t$ is a tuple, we can extract a component of this tuple. After the computation step, the term $t$ may still be available after the computation step. It is still a tuple, so it cannot be used as a signed message.

Since we do not keep the syntactic skeleton of the threads that are built by the context. The context may build two threads which perform a partial interaction of type *migration* with the same markers but with distinct syntactic skeletons.

**Example 6.3.15.** *We consider the following open system:*

$$c^1\langle M\rangle.\mathrm{let}^2\ x = \mathrm{getmessage}(M)\ \mathrm{in}\ \mathrm{let}^3\ y = \mathrm{th}_i^n(M)\ \mathrm{in}\ \overline{\mathrm{test}}^4\langle P\rangle.$$

*This open system first receives a term on an unsafe channel. This term is first matched as a signed message, in case of success the term is then matched as a tuple. It is obvious that no context may force the open system to succeed these two matching.*

*Nevertheless, we give in Fig. 6.13, a computation sequence in our context independent semantics in where these two matching are passed. More precisely:*

1. *At the state $C_0$, the context may build the thread*

   $$(0, t_1, [S_1 \to (ext, t_1), S_2 \to (ext, t_2)]),$$

   *which may compute the partial interaction:*

   $$(output, [S_1; S_2], [], \emptyset, \emptyset).$$

   *this allows the computation of the step $C_0 \longrightarrow C_1$.*

2. *At the state $C_1$, the context may build:*

   - *the thread*
     $$(0, t_2, [S_1 \to (ext, t_3), S_2 \to (ext, t_4)]),$$
     *which may compute the partial interaction:*

     $$(sign, [S_1; S_2], [], \emptyset, \emptyset).$$

$$C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow C_3$$

$$\begin{array}{rcl} C_0 & = & (\{(1,\varepsilon,[c \to (ext,t_1)]\},en,\{t_n \mid n \in \mathbb{N}\})\}) \\ C_1 & = & (\{(2,\varepsilon,[M \to (ext,t_2)]\},en,\{t_n \mid n \in \mathbb{N}\setminus\{1\}\})\}) \\ C_2 & = & (\{(3,\varepsilon,[M \to (ext,t_2)]\},en,\{t_n \mid n \in \mathbb{N}\setminus\{1\}\})\}) \\ C_3 & = & (\{(4,\varepsilon,[M \to (ext,t_2)]\},en,\{t_n \mid n \in \mathbb{N}\setminus\{1\}\})\}) \end{array}$$

Figure 6.13: Trace counter-example in the spi-calculus.

- *and the thread*

$$(0,t_4,[S_1 \to (ext,t_1),S_2 \to (ext,t_1)]),$$

  *which may compute the partial interaction:*

$$(name,[],[],\emptyset,\emptyset).$$

*This allows the computation of the step $C_1 \longrightarrow C_2$.*

3. *At the state $C_2$, the context may build: the thread*

$$(0,t_2,[S_1 \to (ext,t_3),S_2 \to (ext,t_4)]),$$

*which may compute the partial interaction:*

$$(tuple,[S_1;S_2],[],\emptyset,\emptyset).$$

*This allows the computation of the step $C_2 \longrightarrow C_3$.*

#### 6.3.4.2 The case of ambient-like calculi

Our abstraction may loose some constraints on the model structure. For instance, in the case of mobile ambients, the location of an ambient cannot be replaced with any arbitrary ambient location. The intruder must know the full rooting path to reach a given ambient. Then spied data are not necessarily available anywhere in the system. When an untrusted program receives a new name, it may be unable to communicate this name at the top level. These constraints are lost in our context abstraction.

**Example 6.3.16.** *We consider the following open system:*

$$(\nu^{l_1}\mathrm{secret})(\nu^{l_2}\mathrm{a})(\nu^{l_3}\mathrm{b})(\mathrm{a}^1[\mathrm{b}^2[in^3 c.\langle\mathrm{secret}\rangle^4]] \mid c^5[in^6\mathrm{a}.open^7\mathrm{b}])$$

*A safe ambient named* **a** *contains a safe ambient* **b***. The safe ambient* **b** *contains a secret value* **secret***. The ambient* **c** *is unsafe. It has the capability to enter the ambient* **a***. Then, the ambient* **b** *may enter the unsafe ambient* **c***. Then, the ambient* **c** *may get the secret* **secret***. But the context may not root the ambient* **c** *at the top level. Thus, the context may not build threads at top level that uses this secret.*

*Nevertheless, we give in Figs. 6.14 and 6.15, a computation sequence in our context independent semantics when the context may communicate the secret* **secret** *at top level. More precisely:*

1. *At the state $C_0$, the context may build:*
   - *the thread*
     $$\left(0, t_2, \left[\begin{array}{c} S_1 \rightarrow (top, \varepsilon), \\ S_2 \rightarrow (ext, t_3) \end{array}\right]\right),$$
     *which may compute the partial interaction:*
     $$(mov\text{-}ambient, [S_1; S_2], [loc], \emptyset, \emptyset).$$
   - *and the thread*
     $$\left(0, t_3, \left[\begin{array}{c} S_1 \rightarrow (0, t_2); \\ S_2 \rightarrow (ext, t_1) \end{array}\right]\right),$$
     *which may compute the partial interaction:*
     $$(in, [S_1; S_2], [loc], \emptyset, \emptyset).$$

   *This allows the computation of the step $C_0 \longrightarrow C_1$. Then, the context may send threads inside the ambient named* **c***.*

2. *Then, the ambient* **c** *may enter the ambient* **a***. This gives the computation step $C_1 \longrightarrow C_2$.*

3. *Then, the ambient* **b** *may enter the ambient* **c***. This gives the computation step $C_2 \longrightarrow C_3$.*

4. *Then, the ambient* **c** *may open the ambient* **b***. This gives the computation step $C_3 \longrightarrow C_4$.*

5. *At the state $C_4$, the context may build the thread*
   $$(0, t_3, [S_1 \rightarrow (5, \varepsilon)]),$$
   *which may compute the partial interaction:*
   $$(fetch, [S_1], [u], \emptyset, \emptyset).$$
   *This allows the computation of the step $C_4 \longrightarrow C_5$. In $C_5$, the secret is unsafe.*

$$C_0 \longrightarrow C_1 \longrightarrow C_2 \longrightarrow C_3 \xrightarrow{C}{}_4 \xrightarrow{C}{}_5$$

$$C_0 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{bmatrix} loc & \to & (\text{top}, \varepsilon) \\ \text{a} & \to & (l_2, \varepsilon) \end{bmatrix}\right) \\ \left(2, \varepsilon, \begin{bmatrix} loc & \to & (1, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \\ \left(3, \varepsilon, \begin{bmatrix} loc & \to & (2, \varepsilon) \\ c & \to & (ext, t_1) \\ \text{secret} & \to & (l_1, \varepsilon) \end{bmatrix}\right) \\ \left(5, \varepsilon, \begin{bmatrix} loc & \to & (\text{top}, \varepsilon) \\ c & \to & (ext, t_1) \end{bmatrix}\right) \\ \left(6, \varepsilon, \begin{bmatrix} loc & \to & (5, \varepsilon) \\ \text{a} & \to & (l_2, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \end{array} \right\}, en \cup \{(\text{top}, \varepsilon)\}, \{t_n \mid n \in \mathbb{N}\} \right)$$

$$C_1 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{bmatrix} loc & \to & (\text{top}, \varepsilon) \\ \text{a} & \to & (l_2, \varepsilon) \end{bmatrix}\right) \\ \left(2, \varepsilon, \begin{bmatrix} loc & \to & (1, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \\ \left(3, \varepsilon, \begin{bmatrix} loc & \to & (2, \varepsilon) \\ c & \to & (ext, t_1) \\ \text{secret} & \to & (l_1, \varepsilon) \end{bmatrix}\right) \\ \left(5, \varepsilon, \begin{bmatrix} loc & \to & (\text{top}, \varepsilon) \\ c & \to & (ext, t_1) \end{bmatrix}\right) \\ \left(6, \varepsilon, \begin{bmatrix} loc & \to & (5, \varepsilon) \\ \text{a} & \to & (l_2, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \end{array} \right\}, en \cup \left\{ \begin{array}{l} (\text{top}, \varepsilon); \\ (5, \varepsilon) \end{array} \right\}, \{t_n \mid n \in \mathbb{N} \setminus \{3\}\} \right)$$

$$C_2 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \begin{bmatrix} loc & \to & (\text{top}, \varepsilon) \\ \text{a} & \to & (l_2, \varepsilon) \end{bmatrix}\right) \\ \left(2, \varepsilon, \begin{bmatrix} loc & \to & (1, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \\ \left(3, \varepsilon, \begin{bmatrix} loc & \to & (2, \varepsilon) \\ c & \to & (ext, t_1) \\ \text{secret} & \to & (l_1, \varepsilon) \end{bmatrix}\right) \\ \left(5, \varepsilon, \begin{bmatrix} loc & \to & (1, \varepsilon) \\ c & \to & (ext, t_1) \end{bmatrix}\right) \\ \left(7, \varepsilon, \begin{bmatrix} loc & \to & (5, \varepsilon) \\ \text{b} & \to & (l_3, \varepsilon) \end{bmatrix}\right) \end{array} \right\}, en \cup \left\{ \begin{array}{l} (\text{top}, \varepsilon); \\ (5, \varepsilon) \end{array} \right\}, \{t_n \mid n \in \mathbb{N} \setminus \{3\}\} \right)$$

Figure 6.14: Trace counter-example in the ambient calculus.

$$C_3 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (\text{top}, \varepsilon) \\ \text{a} & \rightarrow & (l_2, \varepsilon) \end{array} \right]\right) \\ \left(2, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (5, \varepsilon) \\ \text{b} & \rightarrow & (l_3, \varepsilon) \end{array} \right]\right) \\ \left(4, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (2, \varepsilon) \\ \text{secret} & \rightarrow & (l_1, \varepsilon) \end{array} \right]\right) \\ \left(5, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (1, \varepsilon) \\ c & \rightarrow & (ext, t_1) \end{array} \right]\right) \\ \left(7, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (5, \varepsilon) \\ \text{b} & \rightarrow & (l_3, \varepsilon) \end{array} \right]\right) \end{array} \right\}, en \cup \left\{ \begin{array}{l} (\text{top}, \varepsilon); \\ (5, \varepsilon) \end{array} \right\}, \{t_n \mid n \in \mathbb{N} \setminus \{3\}\} \right)$$

$$C_4 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (\text{top}, \varepsilon) \\ \text{a} & \rightarrow & (l_2, \varepsilon) \end{array} \right]\right) \\ \left(4, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (5, \varepsilon) \\ \text{secret} & \rightarrow & (l_1, \varepsilon) \end{array} \right]\right) \\ \left(5, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (1, \varepsilon) \\ c & \rightarrow & (ext, t_1) \end{array} \right]\right) \end{array} \right\}, en \cup \left\{ \begin{array}{l} (\text{top}, \varepsilon); \\ (5, \varepsilon) \end{array} \right\}, \{t_n \mid n \in \mathbb{N} \setminus \{3\}\} \right)$$

$$C_5 = \left( \left\{ \begin{array}{l} \left(1, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (\text{top}, \varepsilon) \\ \text{a} & \rightarrow & (l_2, \varepsilon) \end{array} \right]\right) \\ \left(5, \varepsilon, \left[ \begin{array}{ccc} loc & \rightarrow & (1, \varepsilon) \\ c & \rightarrow & (ext, t_1) \end{array} \right]\right) \end{array} \right\}, en \cup \left\{ \begin{array}{l} (\text{top}, \varepsilon); \\ (5, \varepsilon); \\ (l_1, \varepsilon) \end{array} \right\}, \{t_n \mid n \in \mathbb{N} \setminus \{3\}\} \right)$$

Figure 6.15: Trace counter-example in the ambient calculus(continued).

## 6.4 Implementing other context abstractions

Other context abstractions can be designed and implemented in our meta-language. For instance, we may design abstraction where all the threads built by the context are kept until they are consumed. This way, the marker of migrating threads and resource may not be released. This allows for a complete abstraction of contexts for the spi-calculus.

Moreover, we could have been more precise about the control the context has over unsafe values. For instance, in the case of the mobile ambients, we could associate each ambient with the set of the values of the trheads that may both be built by the context and be rooted inside this ambient. The implementation of such an abstraction require to index set abstractions by values, which is straightforward.

We did investigate neither completeness results, nor further abstractions, because it would deeply depend on the models we consider.

# Chapter 7

# Abstract Interpretation

Abstract Interpretation is a theory of the approximation of semantics. It formalizes the idea that the semantics may be more or less precise according to the considered level of observation. In static analysis, abstract interpretation is used to derive a decidable semantics from a concrete, non decidable, one. Because of the upper-approximation, the result is not complete: this means that not all the properties of the program are discovered, nevertheless, the result is sound: this means that all the captured properties are satisfied in the concrete semantics.

In this chapter, we introduce a generic abstraction to approximate the behavior of a mobile system. It could indeed apply to any transition system. This abstraction does not depend on the abstracted properties yet: they are left as a parameter of our analysis. Hence, our framework is highly generic, and we can make a reduced product between several analyses. Comparing abstract semantics is usually a very difficult task. We provide local comparison criteria that allows for the comparison of two abstract semantics. We will use this framework to derive an analysis of the control flow in Chap. 8, an analysis of the number of occurrences of the threads in Chap. 9, and an analysis of the number of threads inside computation units in Chap. 10.

## 7.1 Concrete semantics

We denote by $\mathscr{C}$ the set of the non standard configurations that satisfy marker unambiguity (i.e. for any $C \in \mathscr{C}$, for any $(t, id, E), (t', id', E') \in C$, if $[t \sim t'$ and $id = id']$, then $t = t'$ and $E = E'$). We denote by $\Sigma$ the set of the transition labels. The transition labels that we have used contains too much information. We simplify them by removing any information about the continuation choice and about the value substitution. Thus, we make no difference between the label $(\mathscr{R}, (p^k, pi^k, Ct^k), \tau)$ and the label $(\mathscr{R}, (p^k, pi^k))$. We are actually interested in the set $\mathscr{C}(\mathscr{S})$ of all the configurations that a system may take during a finite sequence of computation steps. This is given by its collecting semantics, which is defined in [21]. It can be expressed as the least fixpoint of a $\cup$-complete endomorphism $\mathbb{F}$ on the complete lattice $\wp(\Sigma^* \times \mathscr{C})$ defined as follows:

$$\mathbb{F}(X) = (\{\varepsilon\} \times \mathscr{C}_0^{\text{o}}(\mathscr{S})) \cup \left\{ (u.\lambda, C') \;\middle|\; \exists C \in \mathscr{C},\ (u,C) \in X \textit{ and } C \overset{\lambda}{\hookrightarrow} C' \right\}.$$

## 7.2 Generic abstraction

This least fixpoint is usually not decidable, so we use the Abstract Interpretation framework [23] to compute a sound – but not necessarily complete – approximation of it. More precisely, we use a relaxed version of Abstract Interpretation [24] in which the abstract domain is not supposed to be complete under least upper bound; furthermore, no abstraction function is required.

### 7.2.1 Abstraction definition

**Definition 7.2.1.** An abstraction is a tuple $\mathscr{A} = (\mathscr{C}^\sharp, \sqsubseteq, \sqcup, \bot, \gamma, C_0^\sharp, \rightsquigarrow, \nabla)$ which satisfies:

1. $(\mathscr{C}^\sharp, \sqsubseteq)$ is a pre-order;

2. $\sqcup : \wp_{\text{finite}}(\mathscr{C}^\sharp) \to \mathscr{C}^\sharp$ is such that $\forall A^\sharp \in \wp_{\text{finite}}(\mathscr{C}^\sharp),\ \forall a^\sharp \in A^\sharp,\ a^\sharp \sqsubseteq \sqcup(A^\sharp)$;

3. $\bot \in \mathscr{C}^\sharp$ satisfies $\forall a^\sharp \in \mathscr{C}^\sharp,\ \bot \sqsubseteq a^\sharp$;

4. $\gamma : \mathscr{C}^\sharp \to \wp(\Sigma^* \times \mathscr{C})$ is a monotonic map;

5. $C_0^\sharp \in \mathscr{C}^\sharp$ is such that $\{\varepsilon\} \times \mathscr{C}_0^{\text{o}} \subseteq \gamma(C_0^\sharp)$;

6. $\rightsquigarrow \in \wp(\mathscr{C}^\sharp \times \Sigma \times \mathscr{C}^\sharp)$ is an abstract deterministic labeled transition relation over $\mathscr{C}^\sharp$ such that: $\forall C^\sharp \in \mathscr{C}^\sharp,\ \forall (u,C) \in \gamma(C^\sharp),\ \forall \lambda \in \Sigma,\ \forall \overline{C} \in \mathscr{C}$,

$$C \overset{\lambda}{\hookrightarrow} \overline{C} \Longrightarrow \exists \overline{C}^\sharp \in \mathscr{C}^\sharp,\ (C^\sharp \overset{\lambda}{\rightsquigarrow} \overline{C}^\sharp \textit{ and } (u.\lambda, \overline{C}) \in \gamma(\overline{C}^\sharp));$$

7. $\nabla : \mathscr{C}^\sharp \times \mathscr{C}^\sharp \to \mathscr{C}^\sharp$ is a widening operator:

   - $\forall C_1^\sharp,\ C_2^\sharp \in \mathscr{C}^\sharp,\ C_1^\sharp \sqsubseteq C_1^\sharp \nabla C_2^\sharp \textit{ and } C_2^\sharp \sqsubseteq C_1^\sharp \nabla C_2^\sharp$,

- $\forall (C_n^\sharp)_{n\in\mathbb{N}} \in \left(\mathscr{C}^\sharp\right)^{\mathbb{N}}$, the sequence $(C_n^\nabla)_{n\in\mathbb{N}}$ defined as

$$\begin{cases} C_0^\nabla & = C_0^\sharp \\ C_{n+1}^\nabla & = C_n^\nabla \nabla C_{n+1}^\sharp \end{cases}$$

  is ultimately stationary.

The set $\mathscr{C}^\sharp$ is an abstract domain. It captures the properties we are interested in and abstracts away the other properties. The pre-order $\sqsubseteq$ describes the amount of information which is known about the properties that we approximate. We only use a pre-order to allow some concrete properties to be described by several unrelated abstract elements. The abstract union $\sqcup$ is used to gather the information described by several abstract elements. For the sake of generality, it does not necessarily compute the least upper bound of a finite set of abstract elements (this least bound may not even exist). The abstract element $\perp$ denotes the empty set and it provides the basis for our abstract iteration. The function $\gamma$ is a concretization function which maps each abstract property to the set of the concrete elements which satisfy this property. The abstract element $C_0^\sharp$ describes the properties satisfied by the system initial configurations. The relation $\rightsquigarrow$ is used to mimic the concrete transition system in the abstract domain and the operator $\nabla$ is used to ensure the convergence of the analysis in finitely many iterations.

### 7.2.2 Abstract counterpart

Given an abstraction $\mathscr{A} = (\mathscr{C}^\sharp, \sqsubseteq, \sqcup, \perp, \gamma, C_0^\sharp, \rightsquigarrow, \nabla)$, the abstract counterpart $\mathscr{F}_{\mathscr{A}}^\sharp$ of $\mathbb{F}$ defined as:

$$\mathscr{F}_{\mathscr{A}}^\sharp(C^\sharp) = \sqcup(\{\overline{C}^\sharp \mid \exists \lambda \in \Sigma,\ C^\sharp \overset{\lambda}{\rightsquigarrow} \overline{C}^\sharp\} \cup \{C_0^\sharp\}),$$

satisfies the soundness condition $\forall C^\sharp \in \mathscr{C}^\sharp,\ \mathbb{F} \circ \gamma(C^\sharp) \subseteq \gamma \circ \mathscr{F}_{\mathscr{A}}^\sharp(C^\sharp)$. Since $\mathbb{F}$ is monotonic, $\forall n \in \mathbb{N}, \forall C^\sharp \in \mathscr{C}^\sharp,\ \mathbb{F}^n \circ \gamma(C^\sharp) \subseteq \gamma \circ ({\mathscr{F}_{\mathscr{A}}^\sharp}^n)(C^\sharp)$. On the other hand, since $\mathbb{F}$ is a $\cup$-complete endomorphism, we have $\mathrm{lfp}_\emptyset\ \mathbb{F} = \bigcup\{\mathbb{F}^n(\emptyset) \mid n \in \mathbb{N}\}$. As a consequence, we obtain the soundness of our analysis:

**Theorem 7.2.2.** $\mathrm{lfp}_\emptyset\ \mathbb{F} \subseteq \bigcup\{[\gamma \circ (\mathscr{F}_{\mathscr{A}}^\sharp)^n](\perp) \mid n \in \mathbb{N}\}$.

### 7.2.3 Extrapolated iterates

Following [21], we compute a sound and decidable approximation of our abstract semantics using the widening operator $\nabla$:

**Theorem 7.2.3.** *The abstract iteration [24,25] of $\mathscr{F}^\sharp_{\mathscr{A}}$ defined as follows:*

$$\begin{cases} \mathscr{F}^\nabla_0 & = \bot \\ \mathscr{F}^\nabla_{n+1} & = \begin{cases} \mathscr{F}^\nabla_n & \text{if } \mathscr{F}^\sharp_{\mathscr{A}}(\mathscr{F}^\nabla_n) \sqsubseteq \mathscr{F}^\nabla_n \\ \mathscr{F}^\nabla_n \nabla \mathscr{F}^\sharp_{\mathscr{A}}(\mathscr{F}^\nabla_n) & \text{otherwise} \end{cases} \end{cases}$$

*is ultimately stationary and its limit $[\![\mathscr{S}]\!]_{\mathscr{A}}$ satisfies $\mathscr{C}(\mathscr{S}) \subseteq \gamma([\![\mathscr{S}]\!]_{\mathscr{A}})$.*

The abstract value $[\![\mathscr{S}]\!]_{\mathscr{A}} \in \mathscr{C}^\sharp$ is called the abstract semantics of the mobile system according to the abstraction $\mathscr{A}$.

## 7.3 Abstraction algebra

Our framework is highly extensible. We now give some operations over abstractions to compose them.

### 7.3.1 Cartesian product

**Proposition 7.3.1 (product).** *Let $\mathscr{A}_1 = (\mathscr{C}^\sharp_1, \sqsubseteq_1, \sqcup_1, \bot_1, \gamma_1, C^\sharp_{0_1}, \rightsquigarrow_1, \nabla_1)$ and $\mathscr{A}_2 = (\mathscr{C}^\sharp_2, \sqsubseteq_2, \sqcup_2, \bot_2, \gamma_2, C^\sharp_{0_2}, \rightsquigarrow_2, \nabla_2)$ be two abstractions.*

*The following tuple $(\mathscr{C}^\sharp, \sqsubseteq, \sqcup, \bot, \gamma, C^\sharp_0, \rightsquigarrow, \nabla)$ where:*

- $\mathscr{C}^\sharp = \mathscr{C}^\sharp_1 \times \mathscr{C}^\sharp_2$*;*
- $\sqsubseteq$, $\sqcup$, $\bot$ *and* $\nabla$ *are defined pair-wise;*
- $\gamma : \begin{cases} \mathscr{C}^\sharp \to \wp(\Sigma^* \times \mathscr{C}) \\ (a_1, a_2) \mapsto \gamma_1(a_1) \cap \gamma_2(a_2); \end{cases}$
- $C^\sharp_0 = (C^\sharp_{0_1}, C^\sharp_{0_2})$*;*
- $\rightsquigarrow$ *is defined by:*

  $(a_1, a_2) \overset{\lambda}{\rightsquigarrow} (b_1, b_2)$ *if and only if* $a_1 \overset{\lambda}{\rightsquigarrow}_1 b_1$ *and* $a_2 \overset{\lambda}{\rightsquigarrow}_2 b_2$*;*

*is also an abstraction. We call this abstraction the Cartesian product of the two abstractions $\mathscr{A}_1$ and $\mathscr{A}_2$.*

*Proof.* The tuple $(\mathscr{C}^\sharp, \sqsubseteq, \sqcup, \bot, \gamma, C^\sharp_0, \rightsquigarrow, \nabla)$ satisfies Def. 7.2.1:

- Props. (1),(2),(3),(7) are usual properties of the Cartesian product;

- Prop. (4) is satisfied, since both $\gamma_1$ and $\gamma_2$ are monotonic;
- Prop. (5) holds because we have both $\gamma(C_0^\sharp) = \gamma_1(C_{0_1}^\sharp) \cap \gamma_2(C_{0_2}^\sharp)$ and $\forall i \in \{1;2\}, \{\varepsilon\} \times \mathscr{C}_0^o \subseteq \gamma_i(C_{0_i}^\sharp)$;
- Prop. (6) is satisfied:
  Let $C^\sharp = (a_1, a_2)$ be an abstract element in $\mathscr{C}^\sharp$, $(u,C)$ be a concrete element which satisfies $(u,C) \in \gamma(C^\sharp)$, $\lambda$ be a transition label in $\Sigma$ and $\overline{C}$ a concrete state such that $C \overset{\lambda}{\hookrightarrow} \overline{C}$, we need to construct an abstract element $\overline{C}^\sharp \in \mathscr{C}^\sharp$ such that $C^\sharp \overset{\lambda}{\rightsquigarrow} \overline{C}^\sharp$ and $(u.\lambda, \overline{C}) \in \gamma(\overline{C}^\sharp)$: according to Def. 7.2.1.6, for all $i \in \{1;2\}$, we can choose $b_i \in C_i^\sharp$, such that $a_i \overset{\lambda}{\rightsquigarrow}_i b_i$ and $(u.\lambda, \overline{C}) \in \gamma_i(b_i)$; so, by definition of $\rightsquigarrow$, we have $(a_1, a_2) \overset{\lambda}{\rightsquigarrow} (b_1, b_2)$ and, since $\gamma(b_1, b_2) = \gamma_1(b_1) \cap \gamma_2(b_2)$, we obtain that $(u.\lambda, \overline{C}) \in \gamma(b_1, b_2)$; so $(b_1, b_2)$ is a valid candidate.

□

### 7.3.2 Reduced domain

**Proposition 7.3.2 (Reduction).** *Let $\mathscr{A}_0 = (\mathscr{C}_0^\sharp, \sqsubseteq_0, \sqcup_0, \bot_0, \gamma_0, C_{0_0}^\sharp, \rightsquigarrow_0, \nabla_0)$ be an abstraction, and $\rho$ be a reduction operator*[1] *$\rho : \mathscr{C}_0^\sharp \to \mathscr{C}_0^\sharp$ which satisfies:*

$$\forall a \in \mathscr{C}_0^\sharp,\ \gamma_0(a) \subseteq \gamma_0(\rho(a)).$$

*The following tuple $(\mathscr{C}_\rho^\sharp, \sqsubseteq_\rho, \sqcup_\rho, \bot_\rho, \gamma_\rho, C_{0_\rho}^\sharp, \rightsquigarrow_\rho, \nabla_\rho)$ where*

- $\mathscr{C}_\rho^\sharp = \mathscr{C}_0^\sharp$;
- $\sqsubseteq_\rho = \sqsubseteq_0$;
- $\sqcup_\rho = \sqcup_0$;
- $\bot_\rho = \bot_0$;
- $\gamma_\rho = \gamma_0$;
- $C_{0_\rho}^\sharp = C_{0_0}^\sharp$;
- $\rightsquigarrow_\rho$ *is defined by:*
  *$a \rightsquigarrow_\rho c$ if and only if there exists $b \in \mathscr{C}_0^\sharp$, such that $\rho(a) \rightsquigarrow_0 b$ and $c = \rho(b)$;*

[1] $\rho$ simplifi es the properties obtained in the abstract domain.

- $\nabla_\rho = \nabla_0$;

*is also an abstraction. We call this abstraction the reduction of the abstraction* $\mathscr{A}_0$ *by the reduction operator* $\rho$.

*Proof.* The tuple $(\mathscr{C}^\sharp_\rho, \sqsubseteq_\rho, \sqcup_\rho, \perp_\rho, \gamma_\rho, C^\sharp_{0_\rho}, \rightsquigarrow_\rho, \nabla_\rho)$ satisfies Def. 7.2.1:

- Props. (1),(2),(3),(4),(7) hold because $\mathscr{A}_0$ is an abstraction;

- Prop. (5) is satisfied because:

  - $\{(\varepsilon, c_0) \mid c_0 \in C_0\} \subseteq \gamma_0(C^\sharp_{0_0})$ (since $\mathscr{A}_0$ is an abstraction),
  - $\gamma_0(C^\sharp_{0_0}) \subseteq \gamma_0(\rho(C^\sharp_{0_0}))$ (by definition of $\rho$),
  - $\gamma_0 = \gamma_\rho$ (by definition of $\gamma_\rho$),
  - $C^\sharp_\rho = \rho(C^\sharp_0)$ (by definition of $C^\sharp_{0_\rho}$),

  so $\{\varepsilon\} \times \mathscr{C}^{\mathrm{o}}_0(\mathscr{S}) \subseteq \gamma_0(C^\sharp_{0_0}) \subseteq \gamma_0(\rho(C^\sharp_0)) = \gamma_\rho(C^\sharp_{0_\rho})$;

- Prop. (6) is satisfied:
  Let $a$ be an element of $\mathscr{C}^\sharp_\rho$, $(u,C)$ be a concrete element which satisfies $(u,C) \in \gamma_\rho(a)$, $\lambda$ be a transition label in $\Sigma$ and $\overline{C}$ a concrete state such that $C \overset{\lambda}{\looparrowright} \overline{C}$, we need to construct an abstract element $\overline{b} \in \mathscr{C}^\sharp$ such that $C^\sharp \overset{\lambda}{\rightsquigarrow}_\rho \overline{b}$ and $(u.\lambda, \overline{C}) \in \gamma_\rho(\overline{b})$: we have $(u,C) \in \gamma_0(a) \subseteq \gamma_0(\rho(a))$ and $C \overset{\lambda}{\looparrowright} \overline{C}$, so according to Def. 7.2.1.6, we can choose $b \in \mathscr{C}^\sharp_0$ such that $\rho(a) \overset{\lambda}{\rightsquigarrow}_0 b$ and $(u.\lambda, \overline{C}) \in \gamma(b)$; then, by definition of $\overset{\lambda}{\rightsquigarrow}_\rho$, we have $a \overset{\lambda}{\rightsquigarrow}_\rho \rho(b)$, and, since $\gamma_0(b) \subseteq \gamma_0(\rho(b))$, we obtain that $(u.\lambda, \overline{C}) \in \gamma_0(\rho(b))$; since $\gamma_\rho = \gamma_0$, we have $(u.\lambda, \overline{C}) \in \gamma_\rho(\rho(b))$; so $\rho(b)$ is a valid candidate.

□

Furthermore, $\rho$ may also be used to simplify the final result of the abstract iteration. This way, we refine the abstract semantics of the transition system according to $\mathscr{A}_\rho$, and choose $\rho(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_\rho})$ as new abstract semantics. Unfortunately, because post fixpoint extrapolation is highly non-monotonic, we cannot prove that $\rho(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_\rho}) \sqsubseteq_0 \llbracket \mathscr{S} \rrbracket_{\mathscr{A}_0}$.

## 7.4 Comparing abstractions

Some abstract domains may be related by an approximation: This intuitively means that, starting from the concrete domain, we perform a first abstraction to obtain a first domain, and then we compose another abstraction and obtain the second one. Then, we may be interested in comparing the abstract semantics that is associated with a program in the first domain with the abstract semantics that is associated with the same program in the second domain. Unfortunately, such a comparison is made very difficult because of the use of the widening operator, the behavior of which is highly non monotonic.

Nevertheless we propose some assumptions about the abstractions, so that we can compare their abstract semantics.

### 7.4.1 Monotonic abstraction

The first assumption is that abstraction must be monotonic. This means that when applying any operator, the more precise the arguments are, the more precise the result is. This is formalized in the following definition:

**Definition 7.4.1 (monotonic abstraction).** We say that an abstraction $\mathscr{A} = (\mathscr{C}^\sharp, \sqsubseteq, \sqcup, \bot, \gamma, C_0^\sharp, \rightsquigarrow, \nabla)$ is monotonic if and only if the following assertions are satisfied:

1. 
   - $\forall A, B \in \wp_{\text{finite}}(\mathscr{C}^\sharp)$ such that $A \subseteq B$, we have $\sqcup(A) \sqsubseteq \sqcup(B)$,
   - $\forall A \in \wp_{\text{finite}}(\mathscr{C}^\sharp)$, $\forall a, b \in \mathscr{C}^\sharp$ such that $a \sqsubseteq b$, we have:
   $$\sqcup(A \cup \{a\}) \sqsubseteq \sqcup(A \cup \{b\});$$
2. $\forall a, b, c \in \mathscr{C}^\sharp$, such that $a \sqsubseteq b$ and $a \overset{\lambda}{\rightsquigarrow} c$, there exists $d \in \mathscr{C}^\sharp$ such that $c \sqsubseteq d$ and $b \overset{\lambda}{\rightsquigarrow} d$.
3. the widening operator $\nabla$ is monotonic with respect to each of its arguments.

Roughly speaking, the first assertion describes the monotonicity of the join operator, whereas the second one ensures that the abstract transition relation can simulate itself up-to to information lost. The last assertion is very strong. In practice, it is only satisfied in domains that satisfy the finite chain condition, by using the join operator as widening.

The following properties help in compositionally proving that an abstraction is monotonic:

**Proposition 7.4.2 (monotonicity stability).** *The following properties are satisfied:*

1. *The Cartesian product (see Sect. 7.3.1 on page 160) of two monotonic abstractions is also monotonic.*

2. *The reduction (see Def. 7.3.2 on page 161) of a monotonic abstraction by a monotonic reduction operator is also monotonic.*

### 7.4.2 Local comparison between two abstractions

The second assumption is that we can locally compare the accuracy of the abstractions. This means that the two abstract domains are related with an abstraction function (mapping any abstract property in the more precise domain into an abstract property in the less precise domain). Then, we require that, given an abstract precondition, the abstraction of the result of an abstract computation from this abstract precondition in the more precise domain is always more precise than the result of the corresponding abstract computation in the less precise domain from the abstraction of this abstract precondition. This is formalized in the following definition:

**Definition 7.4.3 (local comparison).** We consider two abstractions $\mathscr{A}_1 = (\mathscr{C}_1^\sharp, \sqsubseteq_1, \sqcup_1, \bot_1, \gamma_1, C_{0_1}^\sharp, \rightsquigarrow_1, \nabla_1)$ and $\mathscr{A}_2 = (\mathscr{C}_2^\sharp, \sqsubseteq_2, \sqcup_2, \bot_2, \gamma_2, C_{0_2}^\sharp, \rightsquigarrow_2, \nabla_2)$. We say that the abstraction $\mathscr{A}_1$ locally approximates the abstraction $\mathscr{A}_2$ if there exists a monotonic concretization function $\alpha^{1\leftarrow 2} \in \mathscr{C}_2^\sharp \rightarrow \mathscr{C}_1^\sharp$ such that:

1. $\forall a \in \mathscr{C}_2^\sharp,\ \gamma_2(a) \subseteq \gamma_1(\alpha^{1\leftarrow 2}(a))$;

2. $\forall A \in \wp_{\text{finite}}(\mathscr{C}_2^\sharp),\ \alpha^{1\leftarrow 2}(\sqcup_2(A)) \sqsubseteq_1 \sqcup_1(\alpha^{1\leftarrow 2}(A))$;

3. $\alpha^{1\leftarrow 2}(\bot_2) \sqsubseteq_1 \bot_1$;

4. $\alpha^{1\leftarrow 2}(C_{0_2}^\sharp) \sqsubseteq_1 C_{0_1}^\sharp$;

5. $\forall a_2, b_2 \in \mathscr{C}_2^\sharp$, such that $a \rightsquigarrow_2 b$, there exists $b_1 \in \mathscr{C}_1^\sharp$ such that $\alpha^{1\leftarrow 2}(a) \rightsquigarrow_1 b_1$ and $\alpha^{1\leftarrow 2}(b_2) \sqsubseteq_1 b_1$;

6. $\forall a_2, b_2 \in \mathscr{C}_2^\sharp$, $\alpha^{1\leftarrow 2}(a_2 \nabla_2 b_2) \sqsubseteq_1 (\alpha^{1\leftarrow 2}(a_2)) \nabla_1 (\alpha^{1\leftarrow 2}(b_2))$.

The following properties help in compositionally proving that an abstraction locally approximates another abstraction:

**Proposition 7.4.4 (local comparison composition).** *The following properties are satisfied:*

1. *If the abstraction $\mathscr{A}_2$ locally approximates the abstraction $\mathscr{A}_1$ and if the abstraction $\mathscr{A}_1$ locally approximates the abstraction $\mathscr{A}_0$, then the abstraction $\mathscr{A}_2$ locally approximates the abstraction $\mathscr{A}_0$.*

2. *The abstraction $\mathscr{A}_1$ locally approximates the Cartesian product (see Sect. 7.3.1 on page 160) of the abstractions $\mathscr{A}_1$ and $\mathscr{A}_2$.*

3. *The abstraction $\mathscr{A}_1$ locally approximates any reduction (see Def. 7.3.2 on page 161) of the abstraction $\mathscr{A}_1$ by any anti-extensive operator.*

4. *If the abstraction $\mathscr{A}_1$ locally approximates the abstraction $\mathscr{A}_2$, then the Cartesian product (see Sect. 7.3.1 on page 160) of the abstractions $\mathscr{A}_1$ and $\mathscr{A}_0$ locally approximates the Cartesian product of the abstractions $\mathscr{A}_2$ and $\mathscr{A}_0$.*

### 7.4.3 Least fixpoint comparison

These two notions are useful to give conditions that ensure that we can compare abstract semantics.

**Theorem 7.4.5 (Abstraction comparison).** *Let us consider two abstractions $\mathscr{A}_1$ and $\mathscr{A}_2$ such that:*

- *the abstraction $\mathscr{A}_1$ is monotonic;*
- *the abstraction $\mathscr{A}_1$ is locally approximating the abstraction $\mathscr{A}_2$ via the concretization function $\alpha^{1\leftarrow 2}$;*

*Then, the abstract semantics of the transition system according to respectively $\mathscr{A}_1$ and $\mathscr{A}_2$ are related by the relation:*

$$\gamma_2(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \subseteq \gamma_1(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1})$$

The proof of Thm. 7.4.5 uses the following Lemma:

**Lemma 7.4.6.** *Let us consider two abstractions $\mathscr{A}_1$ and $\mathscr{A}_2$ that satisfy the assumption of Thm. 7.4.5. Then the abstract semantics of the transition system according to respectively $\mathscr{A}_1$ and $\mathscr{A}_2$ are related by the relation:*

$$\alpha^{1\leftarrow 2}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \sqsubseteq_1 \llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1}.$$

*Proof.*
We denote:

$$\begin{cases} \mathscr{A}_1 = (\mathscr{C}_1^\sharp, \sqsubseteq_1, \sqcup_1, \bot_1, \gamma_1, C_{0_1}^\sharp, \rightsquigarrow_1, \nabla_1) \\ \mathscr{A}_2 = (\mathscr{C}_2^\sharp, \sqsubseteq_2, \sqcup_2, \bot_2, \gamma_2, C_{0_2}^\sharp, \rightsquigarrow_2, \nabla_2) \end{cases}$$

1. we first prove that $\forall a \in \mathscr{C}_2^\sharp,\ \alpha^{1\leftarrow 2}(\mathscr{F}^\sharp_{\mathscr{A}_2}(a)) \sqsubseteq_1 \mathscr{F}^\sharp_{\mathscr{A}_1}(\alpha^{1\leftarrow 2}(a))$:

   Since Def. 7.4.3.(2), we have $\alpha^{1\leftarrow 2}(\mathscr{F}^\sharp_{\mathscr{A}_2}(a)) \sqsubseteq_1 A$, where:

$$A = \sqcup_1(\{\alpha^{1\leftarrow 2}(b) \mid \exists b \in \mathscr{C}_2^\sharp,\ \exists \lambda \in \Sigma,\ a \overset{\lambda}{\rightsquigarrow}_2 b\} \cup \{\alpha^{1\leftarrow 2}(C^\sharp_{0_2})\}).$$

   Then, because of Def. 7.4.3.(5), we know that $\forall b \in \mathscr{C}_2^\sharp$ such that $a \overset{\lambda}{\rightsquigarrow}_2 b$, there exists $b_1 \in \mathscr{C}_1^\sharp$ such that both $\alpha^{1\leftarrow 2}(a) \overset{\lambda}{\rightsquigarrow}_1 b_1$ and $\alpha^{1\leftarrow 2}(b) \sqsubseteq_1 b_1$. Moreover, thanks to Def. 7.4.3.(4), we have $\alpha^{1\leftarrow 2}(C^\sharp_{0_2}) \sqsubseteq_1 C^\sharp_{0_1}$. So, using both properties of Def. 7.4.1.(1), we deduce that:

$$A \sqsubseteq_1 \mathscr{F}^\sharp_{\mathscr{A}_1}(\alpha^{1\leftarrow 2}(a));$$

   And we conclude that:

$$\alpha^{1\leftarrow 2}(\mathscr{F}^\sharp_{\mathscr{A}_2}(a)) \sqsubseteq_1 \mathscr{F}^\sharp_{\mathscr{A}_1}(\alpha^{1\leftarrow 2}(a)). \qquad (7.1)$$

2. Then, we prove that $\mathscr{F}^\sharp_{\mathscr{A}_1}$ is monotonic:
   Let $a, b$ be two abstract elements in $\mathscr{C}_1^\sharp$ such that $a \sqsubseteq_1 b$.

   - By Def. 7.4.1.(2), we know that for any $c$ such that $a \overset{\lambda}{\rightsquigarrow}_1 c$ there exists $d$ such that both $b \overset{\lambda}{\rightsquigarrow}_1 d$ and $c \sqsubseteq_1 d$.
   - We also have $C^\sharp_{0_1} = C^\sharp_{0_1}$.

   So, using Def. 7.4.1.(1), we conclude that:

$$\mathscr{F}^\sharp_{\mathscr{A}_1}(a) \sqsubseteq_1 \mathscr{F}^\sharp_{\mathscr{A}_1}(b).$$

3. We now prove that for any $n \in \mathbb{N}$:

$$\alpha^{1\leftarrow 2}(\mathscr{F}_n^{\nabla_2}) \sqsubseteq_1 \mathscr{F}_n^{\nabla_1}$$

   - if $n = 0$: this is true thanks to Def. 7.4.3.(3).
   - we suppose this is true for $n = n_0$ and we prove it for $n = n_0 + 1$:
     By assumption hypothesis:

$$\alpha^{1\leftarrow 2}\left(\mathscr{F}_{n_0}^{\nabla_2}\right) \sqsubseteq_1 \mathscr{F}_{n_0}^{\nabla_1}.$$

     Then,

$$\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0+1}^{\nabla_2}) \sqsubseteq_1 (\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0}^{\nabla_2}))\nabla_1(\alpha^{1\leftarrow 2}(\mathscr{F}^\sharp_{\mathscr{A}_2}(\mathscr{F}_{n_0}^{\nabla_2}))).$$

– By induction hypothesis,

$$\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0}^{\nabla_2}) \sqsubseteq_1 \mathscr{F}_{n_0}^{\nabla_1}. \tag{7.2}$$

– We now prove that:

$$(\alpha^{1\leftarrow 2}(\mathscr{F}^{\sharp}_{\mathscr{A}_2}(\mathscr{F}_{n_0}^{\nabla_2}))) \sqsubseteq_1 \mathscr{F}^{\sharp}_{\mathscr{A}_1}(\mathscr{F}_{n_0}^{\nabla_1}). \tag{7.3}$$

Since:

∗ because of the result (7.1):

$$(\alpha^{1\leftarrow 2}(\mathscr{F}^{\sharp}_{\mathscr{A}_2}(\mathscr{F}_{n_0}^{\nabla_2}))) \sqsubseteq_1 \mathscr{F}^{\sharp}_{\mathscr{A}_1}(\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0}^{\nabla_2})).$$

∗ and thanks to the monotony of $\mathscr{F}^{\sharp}_{\mathscr{A}_1}$ and to the result (7.2):

$$\mathscr{F}^{\sharp}_{\mathscr{A}_1}(\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0}^{\nabla_2})) \sqsubseteq_1 \mathscr{F}^{\sharp}_{\mathscr{A}_1}(\mathscr{F}_{n_0}^{\nabla_1}).$$

Then, because of the results (7.2) and (7.3), and thanks to the Def. 7.4.1.(3), we can conclude that:

$$(\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0}^{\nabla_2}))\nabla_1(\alpha^{1\leftarrow 2}(\mathscr{F}^{\sharp}_{\mathscr{A}_2}(\mathscr{F}_{n_0}^{\nabla_2}))) \sqsubseteq_1 \mathscr{F}_{n_0+1}^{\nabla_1}.$$

So:

$$\alpha^{1\leftarrow 2}(\mathscr{F}_{n_0+1}^{\nabla_2}) \sqsubseteq_1 \mathscr{F}_{n_0+1}^{\nabla_1}.$$

Moreover, the two sequences $(\alpha^{1\leftarrow 2}(\mathscr{F}_{n}^{\nabla_2}))$ and $(\mathscr{F}_{n_0}^{\nabla_1})$ are ultimately stationary. So their limits $\alpha^{1\leftarrow 2}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2})$ and $\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1}$ satisfy:

$$\alpha^{1\leftarrow 2}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \sqsubseteq_1 \llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1}.$$

□

We now give the proof of Thm. 7.4.5.

*Proof.* By Def. 7.4.3.(1), we have $\gamma_2(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \subseteq \gamma_1(\alpha^{1\leftarrow 2}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}))$. Since $\gamma_1$ is monotonic and by using Lem. 7.4.6, we obtain that $\gamma_1(\alpha^{1\leftarrow 2}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2})) \subseteq \gamma_1(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1})$. By transitivity, we conclude that $\gamma_2(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \subseteq \gamma_1(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1})$. □

Thm. 7.4.5 is useful for comparing the result of different abstractions. Nevertheless, it only applies when the coarse abstraction has a strong structure such as a domain satisfying the ascending chain condition. This restrict the application range of this proposition to the comparison between a very simple analysis in a highly structured domain and a more sophisticated analysis. One application is shown in Chap. 8.

**Corollary 7.4.7.** *If $\mathscr{A}_1$ is monotonic and $\mathscr{A}_2$ is the reduction of the Cartesian product of the abstraction $\mathscr{A}_1$ and an other abstraction by an anti-extensive operator, then $\gamma_2(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_2}) \subseteq \gamma_1(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_1})$.*

# Chapter 8

# Environment approximation

We now use our framework to derive an analysis of the control flow for mobile systems. This analysis requires an abstract domain for describing sets of environments. In Sect. 8.1, we introduce the generic control flow analysis, independently of the abstract domain choice. In Sect. 8.2, we propose a first analysis. This analysis consists in capturing at any program point a relation between the marker of a thread at this program point and the marker of each value in the environment of this thread. This way we generalize the analysis we have already proposed in [32, 36] for the $\pi$-calculus and in [37] for the mobile ambients to any calculus described in our meta language. In Sect. 8.3, we consider a more relational approximation, that may capture comparison between the markers of the values in an environment, even in the case when there is no relationship between the thread marker: this generalizes the framework which is described in [39].

## 8.1 Generic analysis

### 8.1.1 Generic domain

#### 8.1.1.1 Thread abstraction

We assume that we are given a family $(Atom^{\sharp}_{V}, \sqsubseteq_{V}, \sqcup_{V}, \bot_{V}, \nabla_{V})_{V \subseteq \mathscr{V}}$ of abstract domains of properties. For each subset $V \subseteq \mathscr{V}$ of variables, the abstract domain $Atom^{\sharp}_{V}$ is used for globally abstracting some properties about the marker and the environment of a thread the interface of which is the set of variables $V$. The relation $\sqsubseteq_{V}$ is a pre-order which describes the relative amount of information between those properties. For any subset $V \subseteq \mathscr{V}$ of variables, we denote by $Env^{\mathscr{M}}_{V}$ the set $\mathscr{M} \times (V \to (\mathscr{L} \times \mathscr{M}))$ of the pairs that are made of a marker and an environment over $V$. Each abstract property $a \in Atom^{\sharp}_{V}$ is related to $\wp(Env^{\mathscr{M}}_{V})$ by a monotonic concretization function $\gamma_{V}$. The operator $\sqcup_{V}$ maps each finite set of properties to a weaker property: for each finite set $A \subseteq \wp(Atom^\sharp_V)$ and for each element $a \in A$, we have $a \sqsubseteq_V (\sqcup_V A)$. The element $\bot_V$ is the least element in $Atom^\sharp_V$ with respect to $\sqsubseteq_V$. We assume that $\gamma_V$ is strict, that is to say, $\gamma_V(\bot_V) = \emptyset$. The operator $\nabla_V$ is a widening operator. It satisfies $\sqcup_V\{a;b\} \sqsubseteq_V a\nabla_V b$, and for any sequence $(a_n) \in (Atom^\sharp_V)^{\mathbb{N}}$, the sequence $(a^\nabla) \in (Atom^\sharp_V)^{\mathbb{N}}$ that is defined by $a^\nabla_0 = a_0$ and $a^\nabla_{n+1} = a^\nabla_n \nabla_V a_{n+1}$ is ultimately stationary.

Then, our main abstract domain $\mathscr{C}^\sharp_{env}$ is defined as follows:

$$\mathscr{C}^\sharp_{env} = \Pi_{p \in \mathscr{L}_\mathrm{p}} \left( Atom^\sharp_{I_p} \right).$$

The domain structure $(\sqsubseteq_{env}, \sqcup_{env}, \bot_{env}, \nabla_{env})$ is defined component-wise from the family of operator tuples $(\sqsubseteq_V, \sqcup_V, \bot_V, \nabla_V)_{V \subseteq \mathscr{V}}$. The abstract domain $\mathscr{C}^\sharp_{env}$ is related to $\wp(\Sigma^* \times \mathscr{C})$ by the concretization function $\gamma_{env}$ that maps each abstract property $(f_p)_{p \in \mathscr{L}_\mathrm{p}} \in \mathscr{C}^\sharp_{env}$ to the set of configurations $(u, C) \in \Sigma^* \times \mathscr{C}$ such that for any thread $(q, id, E) \in C$, we have $(id, E) \in \gamma_{I(q)}(f_q)$. This way, a configuration $C$ satisfies an abstract properties $f$ if for any thread $(q, id, E)$, the pair $(id, E)$ satisfies the abstraction $(f(q))$ of the threads at program point $q$.

We now introduce some primitives to handle the elements of $Atom^\sharp_V$:

- *initial environment abstraction:* the abstract element $\varepsilon^\sharp_\emptyset$ gives the abstraction of the pair that is made of the initial marker and of an empty environment. This way $\varepsilon^\sharp_\emptyset \in Atom^\sharp_\emptyset$ satisfies:

$$\{(\varepsilon, \emptyset)\} \subseteq \gamma_\emptyset(\varepsilon^\sharp_\emptyset);$$

- *abstract restriction:* the primitive $\nu^\sharp$ describes fresh name allocation. Let $x$ be a variable in $\mathscr{V} \setminus V$, let $l$ be a label in $\mathscr{L}$ and $A$ be an abstract element in $Atom^\sharp_V$. The abstract restriction $\nu^\sharp(x, l, A)$ of $x$ in $A$ is an element in $Atom^\sharp_{V \cup \{x\}}$ which satisfies:

$$\left\{ (id, E) \in Env^{\mathscr{M}}_{V \cup \{x\}} \;\middle|\; \begin{array}{l} (id, E_{|V}) \in \gamma_V(A), \\ E(x) = (l, id), \\ \forall y \in V,\ E(y) \neq (l, id) \end{array} \right\} \subseteq \gamma_{V \cup \{x\}}(\nu^\sharp(x, l, A)).$$

Intuitively, $\nu^\sharp(x, l, A)$ simulates the binding of the variable $x$ to a fresh value. This value is given by associating the marker of the thread with the value label $l$. The concrete semantics ensures that allocated values are fresh. We allow for designing accurate transfer functions, the soundness of which relies on this property.

- *abstract garbage collection:* the primitive $\mathrm{GC}^\sharp$ allows the restriction of environment domains. Let $X$ be a finite subset of $\mathscr{V}$ and $A$ be an element of $Atom^\sharp_V$. The abstract projection, $\mathrm{GC}^\sharp(X,A)$, of $A$ onto $X$ is an abstract element in $Atom^\sharp_{V\cap X}$ which satisfies:

$$\{(id, E_{|V\cap X}) \in Env^{\mathscr{M}}_X \mid (id,E) \in \gamma_V(A)\} \subseteq \gamma_{V\cap X}(\mathrm{GC}^\sharp(X,A)).$$

## 8.1.2 Molecule abstraction

During an interaction, we have to describe the relations among the markers and the environments of several threads. For that purpose, we also assume that we are given a family $(Molecule^\sharp_{(V_i)_i})$ of abstract properties[1]. This family is indexed over the tuples $(V_i)_i \in \wp(\mathscr{V})^*$ of finite subsets of $\mathscr{V}$. For any $n \in \mathbb{N}$ and for any $(V_i)_{1\leq i\leq n} \in \wp(\mathscr{V})^n$, each property in $Molecule^\sharp_{(V_i)_{1\leq i\leq n}}$ is related by a concretization function $\gamma_{(V_i)_{1\leq i\leq n}}$ to the elements of $\wp(\Pi_{1\leq i\leq n}(Env^{\mathscr{M}}_{V_i}))$ which satisfy this property.

We now introduce some primitives to handle the elements of $Molecule^\sharp_{(V_i)}$ for any $(V_i) \in (\wp(\mathscr{V}))^*$ and to relate the families $(Atom^\sharp_V)_{V\subseteq\mathscr{V}}$ and $(Molecule^\sharp_{(V_i)})_{(V_i)\in(\wp(\mathscr{V}))^*}$:

- *abstract injection:* the primitive $\mathrm{INJ}^\sharp$ translates an abstract property about a single thread into an abstract property about an 1-tuple of thread. Let $V \subseteq \mathscr{V}$ be an interface, and let $A$ be an abstract element in $Atom^\sharp_V$. The abstract injection of $A$ is an element in $Molecule^\sharp_{(V)}$ that satisfies:

$$\gamma_V(A) \subseteq \gamma_{(V)}(\mathrm{INJ}^\sharp(A)).$$

- *abstract product:* the primitive $\bullet$ gathers the description of two tuples of threads. Let $m,n$ be the size of the two tuples. Let $(U_i) \in (\wp(\mathscr{V}))^m$ and $(V_i) \in (\wp(\mathscr{V}))^n$ be two tuples of interfaces. We denote by $(W_i) \in (\wp(\mathscr{V}))^{m+n}$ the tuple of interfaces that is defined by $W_i = U_i$ when $1 \leq i \leq m$ and $W_{i+m} = V_i$ when $1 \leq i \leq n$. Let $A$ and $B$ be two abstract elements such that $A \in Molecule^\sharp_{(U_i)_{1\leq i\leq m}}$ and $B \in Molecule^\sharp_{(V_i)_{1\leq i\leq n}}$. The abstract concatenation $A \bullet B$ is an element of $Molecule^\sharp_{(W_i)_{1\leq i\leq m+n}}$ which satisfies:

$$\left\{ (e_i)_{i\in[\![1;m+n]\!]} \;\middle|\; \begin{array}{l} (e_i)_{1\leq i\leq m} \in \gamma_{(U_i)}(A) \\ (e_{i+m})_{1\leq i\leq n} \in \gamma_{(V_i)}(B) \end{array} \right\} \subseteq \gamma_{(W_i)}(A \bullet B).$$

[1]The abstract domain $Molecule^\sharp_{(V_i)_i}$ is not assumed to be a pre-order, because it is only used to make intermediary calculi, and not to make iterations.

For the sake of simplicity, we require that the primitive $\bullet$ is associative, so that we can write $a \bullet b \bullet c$ instead of $(a \bullet b) \bullet c$.

- *abstract projections:* the primitive $\text{PROJ}^\sharp$ extracts the description of a thread from the description of a tuple of threads. Let $n$ be an integer. Let $(V_i) \in (\wp(\mathcal{V}))^n$ be an $n$-tuple of interfaces. Let $A \in \mathit{Molecule}^\sharp_{(V_i)_{1\leq i\leq n}}$ be an abstract element. Let $k$ be an integer such that $k \leq n$. The abstract projection $\text{PROJ}^\sharp(k,A)$ of the abstract element $A$ onto its $k$-th component is an element of $\mathit{Atom}^\sharp_{V_k}$ that satisfies:

$$\left\{(id_k,E_k) \;\middle|\; \exists\,(id_i,E_i)_i \in \gamma_{(V_i)_i}(A)\right\} \subseteq \gamma_{V_k}(\text{PROJ}^\sharp(k,A));$$

- *abstract extension:* the primitive $\text{NEW}^\sharp_\top$ extends environment domains: given a subset of variables, it erases any information about the variables that already exist and creates the variables that are missing with no information about their value.

  Let $n$ be an integer. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces, let $X$ be a subset of $\mathcal{V} \times [\![1;n]\!]$ and let $A$ be an abstract element in $\mathit{Molecule}^\sharp_{(V_i)}$. For each $i \in [\![1;n]\!]$, we define the set $U_i \subseteq \mathcal{V}$ of variables as $V_i \setminus \{x \mid (x,i) \in X\}$ and the set $W_i \subseteq \mathcal{V}$ of variables by $V_i \cup \{x \mid (x,i) \in X\}$. For each $i$ such that $1 \leq i \leq n$, the set $U_i$ is the set of the variables of the $i$-th interface that are kept unchanged whereas the set $W_i$ is the set of the variables of the $i$-th interface after the abstract extension. The abstract extension $\text{NEW}^\sharp_\top(X,A)$ of $A$ by $X$ is an element in $\mathit{Molecule}^\sharp_{(W_i)}$ which satisfies:

$$\left\{(id_i,E_i) \in \Pi(\mathit{Env}^{\mathcal{M}}_{W_i}) \;\middle|\; \begin{array}{l} \exists(id_i,E'_i) \in \gamma_{(V_i)}(A), \\ \forall i \in [\![1;n]\!], \forall x \in U_i, \\ E'_i(x) = E_i(x) \end{array}\right\} \subseteq \gamma_{(W_i)}(\text{NEW}^\sharp_\top(X,A)).$$

- *abstract synchronization:* the primitive $\text{SYNC}^\sharp$ enforces synchronization conditions between both the values of the variables of some threads and the identities of these threads. In order to compute the identity value of each thread, this primitive also requires the tuple of the program point labels of the threads (since the identity of a thread is given by a pair which is made of the program point label and the thread marker).

  Let $n \in \mathbb{N}$ be an integer. Let $A$ be an abstract element in $\mathit{Molecule}^\sharp_{(V_i)_{1\leq i\leq n}}$, let $(p_i) \in \mathcal{L}_{\text{p}}^n$ be a tuple of program point labels and $S$ be a set of constraints[2]

[2] We recall that the variable $(I,k)$ denotes the $k$-th thread identity.

of the form $(x,k) \diamond (y,l)$ where $\diamond \in \{=;\neq\}$, $k,l \in [\![1;n]\!]$, $x \in V_k \cup \{I\}$, and $y \in V_l \cup \{I\}$, the abstract synchronization $\textsc{sync}^\sharp(S,(p_i),A)$ of $A$ according to the set of constraints $S$ is an element of $\mathit{Molecule}^\sharp_{(V_i)}$ such that:

$$\{(id_i,E_i) \in \gamma_{(V_i)}(A) \mid \forall (a \diamond b) \in S,\ \rho(a) \diamond \rho(b)\} \subseteq \gamma_{(V_i)}(\textsc{sync}^\sharp(S,(p_i),A)),$$

where $\rho((x,i)) = E_i(x)$ when $x \in V_i$ and $\rho((I,i)) = (p_i, id_i)$.

- *abstract marker allocation:* the primitive $\textsc{fetch}^\sharp$ updates the marker of the first thread, according to thread marker allocation scheme. Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program point labels, and $A$ be an element of $\mathit{Molecule}^\sharp_{(V_i)}$, the abstract marker allocation $\textsc{fetch}^\sharp((p_i),A)$ is in $\mathit{Molecule}^\sharp_{(V_i)}$ such that:

$$\left\{ (\overline{id}_i, E_i) \;\middle|\; \begin{array}{l} \forall i \in [\![1;n]\!],\ (id_i,E_i) \in \gamma_{(V_i)}(A) \\ id_i \neq \overline{id}_1 \\ \forall x \in V_i, y \in \mathscr{L},\ (y,\overline{id}_1) \neq E_i(x) \end{array} \right\} \subseteq \gamma_{(V_i)}(\textsc{fetch}^\sharp((p_i),A))$$

where $\overline{id}_i = \begin{cases} N((p_i)_{1 \leq i \leq n}, id_1, \ldots, id_n) & \text{if } i = 1 \\ id_i & \text{otherwise.} \end{cases}$

As in the definition of the restriction, we take into account the fact that the allocated marker is fresh. This helps in designing precise transfer functions the soundness of which relies on this property.

### 8.1.3 Abstract operational semantics

We now use these generic primitives to simulate in the abstract the concrete operational semantics.

#### 8.1.3.1 Exhibited action

In the abstract, each program point label is associated with a property that describes a set of threads. A guard that constraints one thread may give information about the other threads via information closure (or transitive closure). Thus it may not be precise to isolate single threads when matching guards in the abstract. That is why we delay the checking of guard matching, so that we can consider all synchronization constraints at once which allows a better closure of information.

Thus given a program point label $p$ and a partial interaction $pi$. We say that the partial interaction $pi$ may be performed at a program point labeled with $p$ if and only if $pi \in \mathit{interaction}(p)$.

#### 8.1.3.2 Reactive molecule

In the concrete, a reduction step is enabled if each thread satisfies the guard of the partial interaction that it computes and if the synchronization constraints of the rule are satisfied. In the abstract, all these constraints are checked at the same time.

Given an $n$-tuple of program point labels, an $n$-tuple of parameter sequences, an $n$-tuple of matching constraint sets, a set of synchronization constraints and an abstract element $C \in \mathscr{C}^{\sharp}_{env}$, we can compute the abstraction of all the tuples of threads at the corresponding program points that may synchronize their computation by satisfying both their matching guards and the synchronization constraints of the rule. Whenever the result is the bottom element, the synchronization is not possible in the concrete, so the abstract computation step is disabled, otherwise we can refine the abstract property by taking into account the synchronization constraints.

We define a primitive $reagents^{\sharp}$ that takes an $n$-tuple of program point labels, an $n$-tuple of parameter sequences, an $n$-tuple of matching guard constraint sets, a set of synchronization constraints and an abstract property in $\mathscr{C}^{\sharp}_{env}$. The primitive $reagents^{\sharp}$ returns the abstraction of the set of the $n$-tuple of the threads such that each thread satisfies the abstract property and such that these threads may synchronize their computation to compute the rule.

**Definition 8.1.1 (abstract reactive molecule).** Let $n$ be an integer. Let $(p^k)_{1\leq k\leq n}$ be an $n$-tuple of program point labels. Let $(param^k_l)_{k,l}$ be an $n$-tuple of parameter sequences ($param^k_l$ is associated with the $l$-th parameter of the $k$-th thread). Let $(constraints^k)_{1\leq k\leq n}$ be an $n$-tuple of matching guard constraint sets. Let *compatibility* be a set of global synchronization constraints. Let $C^{\sharp} \in \mathscr{C}^{\sharp}_{env}$ be an abstract element. We define the abstract element $reagents^{\sharp}((p^k),(param^k_l)_{k,l},(constraints^k)_k,compatibility,C^{\sharp}) \in Molecule^{\sharp}_{(I(p^k))_{1\leq k\leq n}}$ as:

$$\textsc{sync}^{\sharp}(cons,(p^k),mol),$$

where:

- $mol \triangleq \textsc{inj}^{\sharp}(C^{\sharp}(p_1)) \bullet \ldots \bullet \textsc{inj}^{\sharp}(C^{\sharp}(p_n))$;
- $cons \triangleq \bigcup(\{R_k \mid 0 \leq k \leq n\})$;
- $R_0 \triangleq \{\sigma(X) = \sigma(Y) \mid (X,Y) \in compatibility\}$,

  with $\sigma : \begin{cases} I^k \mapsto (I,k) \\ X^k_l \mapsto (param^k_l,k); \end{cases}$

- $\forall k \in [\![1;n]\!],\ R_k = \{(x,k) \diamond (y,k) \mid x \diamond y \in \mathit{constraints}^k\}$;

In this definition, the set $R_0$ contains the constraints that come from the formal rule whereas each set $R_k$ with $1 \leq k \leq n$ contains the constraints that are due to the matching guard of the $k$-th thread. We use the parameters $(\mathit{param}_l^k)$ so that these constraints are about the variable of the threads instead of the formal variables that are used in the description of the rule.

#### 8.1.3.3 Marker computation and value passing

In the abstract, we pass values and we compute thread markers simultaneously. The reason is that both value passing and marker computation overwrite the old values and the old markers. These old values and these old markers may allow a better accuracy thanks to some information closure. Thus performing marker computation before value passing, or the opposite may lead to information loss.

Value passing and thread marker computation are performed in several steps. First we declare ghost variables that will be associated with passed values without overwriting the old values. Then we simulate marker computation. At this point, we can forget about the former value of the variables. Last, we simulate value passing by synchronizing each ghost variable with the variable that is actually bound.

**Definition 8.1.2 (abstract value passing and marker computation).** Let $t$ be a partial interaction name type in $\{\mathit{replication};\mathit{computation};\mathit{migration}\}$, let $n$ be an integer, let $(p^k)_{1\leq k\leq n}$ be an $n$-tuple of program point labels, let $(\mathit{bd}_l^k)_{k,l}$ be an $n$-tuple of sequences of variables ($\mathit{bd}_l^k$ is associated with the $l$-th variable in the $k$-th thread by the interaction). We denote by $\mathscr{V}_{rd}$ the set of the pairs $(k,l)$ such that $\mathit{bd}_l^k$ is well-defined. Let $(\mathit{param}_l^k)_{k,l}$ be an $n$-tuple of sequences of parameters ($\mathit{param}_l^k$ is associated with the $l$-th parameter of the $k$-th thread). Let $v$-$\mathit{passing}$ be a partial map from $\mathscr{V}_f^Y$ into $\mathscr{V}_f^X \cup \mathscr{V}_f^I$. Let $\mathit{molecule}^\sharp \in \mathit{Molecule}^\sharp_{(I(p^i))_{1\leq i\leq n}}$ be the abstraction of $n$ interacting threads. Let $(Z_l)_{l\in\mathbb{N}}$ be a family of fresh distinct ghost variables in $\mathscr{V}$.

First we declare ghost variables. So we define the molecule $C_1$ as follows:

$$C_1 \triangleq \mathrm{NEW}^\sharp_\top(\{(Z_l,k) \mid (k,l) \in \mathscr{V}_{rd}\}, \mathit{molecule}^\sharp).$$

Then we simulate value passing, by associating each ghost variables with the value that will be passed to their real corresponding variable. This way, we define the molecule $C_2$ as follows:

$$C_2 \triangleq \mathrm{SYNC}^\sharp(\mathit{cons}_1 \cup \mathit{cons}_2, (p^k), C_1),$$

where:

- $cons_1 = \{(Z_l,k) = (param_{l'}^{k'},k') \mid \exists k,k',l,l' \in \mathbb{N},\ v\text{-}passing(Y_l^k) = X_{l'}^{k'}\}$,
- $cons_2 = \{(Z_l,k) = (I,k') \mid \exists k,k',l \in \mathbb{N},\ v\text{-}passing(Y_l^k) = I^{k'}\}$,

Then we simulate the computation of the thread marker: whenever the first partial interaction is not a replication, markers are left unchanged; otherwise we use the primitive $\text{FETCH}^\sharp$. We define the molecule $C_3$ as follows:

$$C_3 \overset{\Delta}{=} \begin{cases} \text{FETCH}^\sharp((p^k),C_2) & \text{if t} = \mathit{replication} \\ C_2 & \text{otherwise.} \end{cases}$$

The next step consists in removing any information about the real variables that receive another value. We define the molecule $C_4$ as follows:

$$C_4 \overset{\Delta}{=} \text{NEW}^\sharp_\top(\{(bd_l^k,k) \mid (k,l) \in \mathscr{V}_{rd}\},C_3).$$

The last step consists in synchronizing each ghost variable with its corresponding real variable. For that purpose, we define the molecule $\mathit{marker\text{-}value}(t,(p^k)_k,\mathit{molecule}^\sharp,(bd_l^k)_{k,l},(\mathit{param}_l^k)_{k,l},v\text{-}passing)$ as:

$$\text{SYNC}^\sharp(\{(Z_l,k) = (bd_l^k,k) \mid (k,l) \in \mathscr{V}_{rd}\},(p^k),C_4).$$

#### 8.1.3.4 Launching a continuation

Once value passing and thread marker allocation have been simulated in the abstract, we can split the abstract molecule and launch continuations.

First we introduce a primitive $\mathit{update}^\sharp$. This primitive takes an abstract thread $A \in \mathit{Atom}_V^\sharp$ and a static environment $E_s \in V_s \to \mathscr{L}$ and extends the environment of the abstract thread in order to simulate in the abstract the creation of the fresh values. First this primitive removes any information about the freshly bound variables, then it uses the primitive $\nu^\sharp$ to map each variable in the domain of the static environment $E_s$ with a sound abstract value. Since the result may depend on the order on which variable are declared, we suppose that we are given a total order $\leq_\mathscr{V}$ over the set of variables.

**Definition 8.1.3 (abstract environment updating).** Let $E_s$ be a static environment over $V_s$ (i.e. $E_s \in V_s \to \mathscr{L}$). Since $(\mathscr{V},\leq_\mathscr{V})$ is a total order, we can write $Dom(E_s) = \{x_i \mid 1 \leq i \leq n\}$ with $x_1 \leq_\mathscr{V} \ldots \leq_\mathscr{V} x_n$. Let $A \in \mathit{Atom}_V^\sharp$ be an abstract thread. We define the abstract thread $\mathit{update}^\sharp(E_s,A) \in \mathit{Atom}^\sharp_{V\cup\{Dom(E_s)\}}$ as follows:

$$\mathit{update}^\sharp(E_s,C) \overset{\Delta}{=} C_n,$$

where:

- $C_0 \overset{\Delta}{=} \text{GC}^\sharp(V \setminus \{Dom(E_s)\}, C)$,
- $C_{k+1} \overset{\Delta}{=} v^\sharp(x_{k+1}, E_s(x_{k+1}), C_k)$, $\forall k \in [\![0;n[\![$.

We can now introduce the primitive *launch*$^\sharp$ which describes the launching of the continuations of the partial interactions. In the abstract, we consider any choice of potential continuations at once. The primitive *launch*$^\sharp$ requires the syntactic continuation of the partial interaction and the abstraction of the system state. It computes the abstraction of the state of the system just after having launched any of the potential continuations: for any potential continuation, first we extend the environment in the abstract by using the primitive *update*$^\sharp$, then we restrict the environment to the new interface by using the abstract garbage collector $\text{GC}^\sharp$. Then we gather all the potential results.

**Definition 8.1.4 (abstract continuation launching).** Let $n$ be an integer. Let $(V_i)$ be an $n$-tuple of interfaces. Let $mol \in Molecule^\sharp_{(V_i)_{1\leq i\leq n}}$ be an abstraction of an $n$-tuple of threads. Let $(ct^k) \in \wp(\wp(\mathscr{L}_\text{p} \times (\mathscr{V} \rightharpoonup \mathscr{L})))^n$ be an $n$-tuple of set of potential continuations. We define the abstract element $launch^\sharp((p^k, ct^k), mol)$ in $\mathscr{C}^\sharp_{env}$ as:

$$\left[p' \mapsto \sqcup_{I(p')} \left\{ \text{GC}^\sharp(I(p'), update^\sharp(E_s, \text{PROJ}^\sharp(k', mol))) \;\middle|\; \exists k',\ (p', E_s) \in \bigcup ct_{k'} \right\}\right].$$

### 8.1.3.5 Broadcast value passing

Broadcast value passing consists in substituting all the occurrences of a value in the system by another one. In the abstract, a variable is associated with an abstract set of values. So we carry out this broadcast substitution by partitioning the threads according to the value that is associated to each variable: we denote by *broadcast* the broadcast substitution (it is a partial map from $\mathscr{V}^I_\mathscr{R}$ into $\mathscr{V}^X_\mathscr{R} \cup \mathscr{V}^I_\mathscr{R}$). For each program point $p$, we will partition the abstract threads at program point $p$ according to which elements of the domain of *broadcast* match the values of the variables in the environments of the threads. For each variable in $I(p)$, there are two possibilities: either the value of the variable is not replaced by the broadcast substitution, or the value of the variable is replaced with the value of a formal variable $broadcast(X)$ where $X \in Dom(broadcast)$. In order to carry out more relational information, we replace all the variables of a thread at once. Thus we partition the abstract threads at program point $p$ in $(card(Dom(broadcast)) + 1)^{card(I(p))})$ classes. Each class is denoted by a mapping $\rho \in (I(p) \to \{0\} \cup Dom(broadcast))$. The association $\rho(x) = 0$ means that the substitution does not change the value of the variable $x$ whereas the association $\rho(x) = I^k$ means that the value of the variable $x$ is equal to the identity the $k$-th interacting thread. For each class, we compute the substitution in several steps. First we restrict the abstract thread so that it only describes concrete threads that satisfy the matching conditions expressed by $\rho$. Then we simulate the substitution in the abstract: for that purpose, we associate each modified variable with a ghost variable, we bind this ghost variable with the new value (this prevents loosing relational information), we forget any information about modified variables and then synchronize them with their ghost twin.

**Definition 8.1.5.** Let $C$ be an abstract element in $\mathscr{C}^{\sharp}_{env}$. Let $n$ be an integer. Let $(p^k)$ be an $n$-tuple of program points. Let $molecule^{\sharp}$ be an element of $Molecule^{\sharp}_{(I(p^k))_k}$. Let $(param^k_l)_{k,l}$ be an $n$-tuple of parameter sequences. Let *broadcast* be a partial map from $\mathscr{V}^I_{\mathscr{R}}$ into $\mathscr{V}^X_{\mathscr{R}} \cup \mathscr{V}^I_{\mathscr{R}}$. Let $q$ be a program point and $\rho$ be a function from $I(q)$ into $Dom(broadcast) \cup \{0\}$. We introduce the set $\mathscr{V}_{mod}$ of the variable that changes their values as $\{a \in I(p) \mid \rho(a) \neq 0\}$. We introduce the set of fresh variables $\mathscr{V}_{ghost} = \{\overline{a} \mid a \in \mathscr{V}_{mod}\}$.

We want to compute the abstraction of the threads at program point $q$ obtained by applying a substitution according to $\rho$ on each thread that satisfies the abstract property $C(q)$. First we gather the abstraction of the interacting threads *molecule* and the abstraction of the threads at program point $q$. We introduce the abstract molecule $C_1 \in Molecule^{\sharp}_{(I(p^1),\ldots,I(p^n),I(q))}$ by:

$$C_1 \stackrel{\Delta}{=} molecule^{\sharp} \bullet \text{INJ}^{\sharp}(C(q)).$$

Then we create the ghost variables. For that purpose, we define the abstract molecule $C_2 \in Molecule^{\sharp}_{(I(p^1),\ldots,I(p^n),(I(q)\cup\mathscr{V}_{ghost}))}$ as follows:

$$C_2 \stackrel{\Delta}{=} \text{NEW}^{\sharp}_{\top}(\{(\overline{a},n+1) \mid a \in \mathscr{V}_{ghost}\},C_1).$$

Then we takes into account $\rho$ and restrict the abstract properties so that it abstracts the threads that satisfy the assumptions that are described by $\rho$ only. For each variable $a \in I(q)$, we introduce $R(a)$ as the set of constraints that are related to $a$ as follows: in the case when $\rho(a) = 0$, the value associated with $a$ matches no variable in the domain of *broadcast* — this way we define $R(a) \stackrel{\Delta}{=} \{(a,n+1) \neq (I,k) \mid \forall I^k \in Dom(broadcast)\}$; in the case when $\rho(a) = I^k$, the value associated with $a$ necessarily matches the value associated with the formal variable $I^k$, thus we set $R(a) \stackrel{\Delta}{=} \{(a,n+1) = (I,k)\}$. Thus we define $C_3 \in Molecule^{\sharp}_{(I(p^1),\ldots,I(p^n),(I(q)\cup\mathscr{V}_{ghost}))}$ as follows:

$$C_3 \stackrel{\Delta}{=} \text{SYNC}^{\sharp}(\cup\{R(a) \mid a \in I(q)\},(p^1,\ldots,p^n,q),C_2).$$

Then we simulate the broadcast substitution over the ghost variables. We define $C_4 \in \mathit{Molecule}^\sharp_{(I(p^1),\dots,I(p^n),(I(q)\cup\mathcal{V}_{ghost}))}$ as follows:

$$C_4 \stackrel{\Delta}{=} \text{SYNC}^\sharp(com,(p^1,\dots,p^n,q),C_3),$$

where $com = \{(\overline{a},n+1) = \sigma(\rho(a)) \mid a \in \mathcal{V}_{mod}\}$ and $\sigma : \begin{cases} I^k \mapsto (I,k) \\ X^k_l \mapsto (param^k_l,k). \end{cases}$
We can now forget any information about the variables that change their values. We define $C_5 \in \mathit{Molecule}^\sharp_{(I(p^1),\dots,I(p^n),(I(q)\cup\mathcal{V}_{ghost}))}$ as follows:

$$C_5 \stackrel{\Delta}{=} \text{NEW}^\sharp_\top(\mathcal{V}_{mod},C_4).$$

Then we copy the information about the ghost variables to their twin variables. We define $C_6 \in \mathit{Molecule}^\sharp_{(I(p^1),\dots,I(p^n),(I(q)\cup\mathcal{V}_{ghost}))}$ as follows:

$$C_6 \stackrel{\Delta}{=} \text{SYNC}^\sharp(\{(a,n+1) = (\overline{a},n+1) \mid a \in \mathcal{V}_{mod}\},(p^1,\dots,p^n,q),C_5).$$

Then we extract information about threads at program point $q$ and define $aux^\sharp(C,molecule^\sharp,(param^k_l),broadcast)(q,\rho)$ in $\mathit{Atom}^\sharp_{I(q)}$ as:

$$\text{GC}^\sharp(I(q),\text{PROJ}^\sharp(n+1,C_6))$$

We now define the primitive $glob^\sharp$ that takes a binding between formal variables and the parameters of the threads that interact, a substitution among formal variables and a set of threads and applies the substitution in the environment with all these threads:

**Definition 8.1.6 (abstract broadcast value passing).** Let $n$ be an integer. Let $(p^k)$ be an $n$-tuple of program points. Let $molecule^\sharp$ be an element of $\mathit{Molecule}^\sharp_{(I(p^k))_k}$. Let $(param^k_l)_{k,l}$ be an $n$-tuple of parameter sequences. Let $broadcast$ be a partial map from $\mathcal{V}^I_{\mathcal{R}}$ into $\mathcal{V}^X_{\mathcal{R}} \cup \mathcal{V}^I_{\mathcal{R}}$. Let $C$ be an abstract element in $\mathcal{C}^\sharp_{env}$. We define the element $glob^\sharp(molecule^\sharp,(param^k_l),broadcast,C)$ as:

$$\left[q \mapsto \sqcup_{I(q)}\{aux^\sharp(C,molecule^\sharp,(param^k_l),broadcast)(q,\rho) \mid \rho \in choice(q)\}\right],$$

where $choice(q) = (I(q) \to \{0\} \cup Dom(broadcast))$.

Let $C^\sharp \in \mathscr{C}^\sharp_{env}$ be an abstract configuration.
Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule.
Let $(p^k)_{1\leq k\leq n} \in \mathscr{L}_\mathrm{p}^n$ be an $n$-tuple of program point labels and $(pi^k)_{1\leq k\leq n} = (s^k, (parameter^k_l)_l, (bd^k_l)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$ be an $n$-tuple of partial interactions.
We introduce:

$$mol \stackrel{\Delta}{=} reagents^\sharp((p^k), (parameter^k_l)_{k,l}, constraints^k, compatibility, C^\sharp).$$

If:

1. $\forall k \in [\![1;n]\!],\ pi^k \in interaction(p)$;
2. $mol \neq \bot_{(I(p_k))_{1\leq k\leq n}}$,

then:

$$C \overset{(\mathscr{R},(p^k,pi^k)_k)}{\longrightarrow_{env}} subs^\sharp(mol, \sqcup_{env}\{C; new\}),$$

where:

- $mol' = marker\text{-}value(type(s^1), (p^k), mol, (bd^k_l), (parameter^k_l), v\text{-}passing)$;
- $new = launch^\sharp((p^k, continuation^k)_k, mol')$;
- $subs^\sharp(molecule, D) = glob^\sharp(molecule, \left(parameter^k_l\right)_{k,l}, broadcast, D)$.

Figure 8.1: Abstract semantics.

### 8.1.4 Abstract operational semantics

We use these primitives in order to describe both the abstraction of the initial states and the abstract computation rule. The abstraction of the initial states is obtained by applying the primitive $launch^\sharp$ with the potential continuations in $init_s$ and the abstract initial environment. Thus the abstraction $C_{0_{env}}$ of the initial states is defined as:

$$C_{0_{env}} = launch^\sharp(init_s, \varepsilon^\sharp_\emptyset).$$

Computation steps are described by the abstract reduction relation in Fig. 8.1. We recall the different steps of this computation, as follows:

- *interaction enabling*:
  - first, we must find some threads that exhibit the right partial interactions;
  - then we check that their interface are compatible with both formal rule synchronization constraints and each matching constraint set;
- *interaction computation*:
  - we abstract marker allocation and value passing;
  - we launch in parallel any potential continuation;
  - we apply broadcast substitution with the whole system in order to model potential re-addressing.

**Theorem 8.1.7.** $(\mathscr{C}^\sharp_{env}, \sqsubseteq_{env}, \sqcup_{env}, \bot_{env}, \gamma_{env}, C_{0_{env}}, \rightarrow_{env}, \nabla_{env})$ *is an abstraction.*

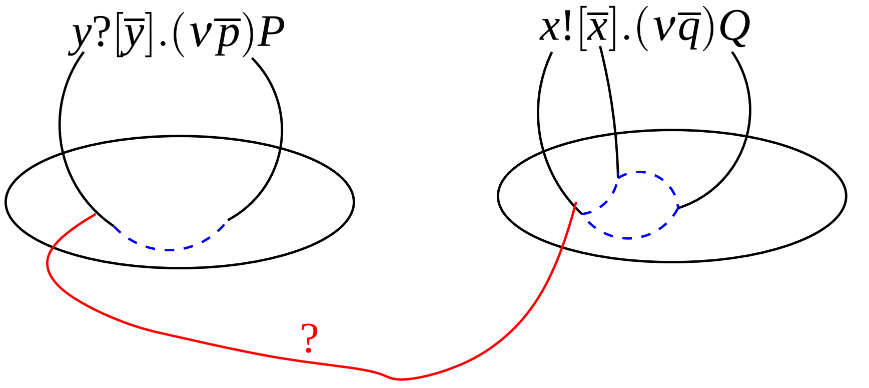
The proof of Thm. 8.1.7 is shown in appendix D.1.

**Example 8.1.8 (communication in the $\pi$-calculus).** *We describe in Fig. 8.2 the abstract communication between two threads in the $\pi$-calculus. We consider two partial interactions: an input and an output. We have some information about the marker and the environment of the threads that compute these partial interactions. This information is described by an ovoid. Dotted lines describe relational information between variable values. Outgoing edges describe information about a particular variable. First, in Fig. 8.2(a), we check that the communication is possible by testing whether the two variables on which the input and the output are performed may be bound to the same value. In such a case, the communication is enabled. In Fig. 8.2(b), we extend the environment of the threads to take into account new variables. We take into account some constraints: synchronization constraints, the fact that the variables in the input are now associated with the communicated values, and the fact that the marker of a fresh value is the marker* *of the thread that has declared this value. In Fig. 8.2(c), we describe information closure: it uses the constraints about the two thread descriptions to collect constraints about the marker and the environment of each thread. At this point, the two descriptions may be separated.*

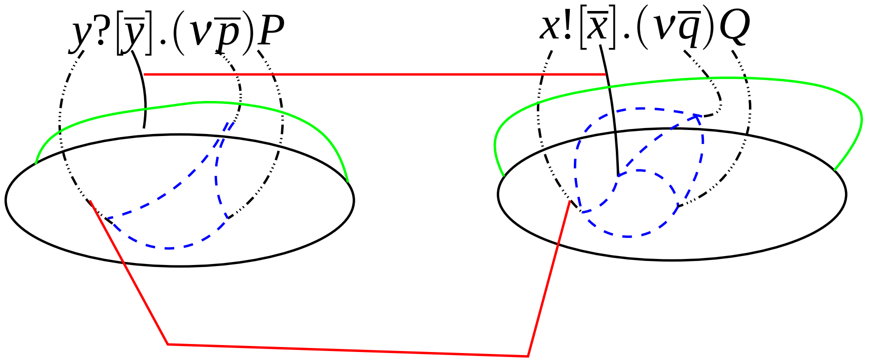
**Example 8.1.9 (migration in the ambient calculus).** *We describe in Fig. 8.3 the abstract migration inside an ambient. We consider three partial interactions: two moving ambients and a capability to enter inside an ambient. We have some information about the marker and the environment of the threads that compute these partial interaction. This information is described by an ovoid. Dotted lines describe relational information between variable values. Outgoing edges describe information about a particular variable, the outgoing edges at the top left of each partial interaction denote information about the location of the thread that computes this partial interaction, the outgoing edges that comes from ambient boundaries denote information about the identity of the ambients. First, in Fig. 8.3(a), we check that the migration is possible by testing whether the three constraints — the first ambient may be inside the second, the capability to enter is inside the first ambient, and the capability provides capability to enter in the second ambient — are simultaneously satisfiable. In such a case, the migration is enabled. In Fig. 8.3(b), we extend the environment of the capability thread to take into account new variables. The location of the migrating ambient is redefined (it is unplugged from the relational ovoid), the old location is described by a pending edge: it is still useful to describe relational information. We take into account some constraints: the preconditions constraints, the fact that the first ambient location is now synchronized with the second ambient identity and the fact that the marker of a fresh value is the marker of the thread that has declared this value. We also describe information closure: it uses the constraints about the two thread descriptions to collect constraints about the marker and the environment of each thread. At this point, the two descriptions may be separated. Then we perform information closure, by following relational information dotted path and separate the description of the pair marker/environment of each thread.*

## 8.2 Control flow analyses

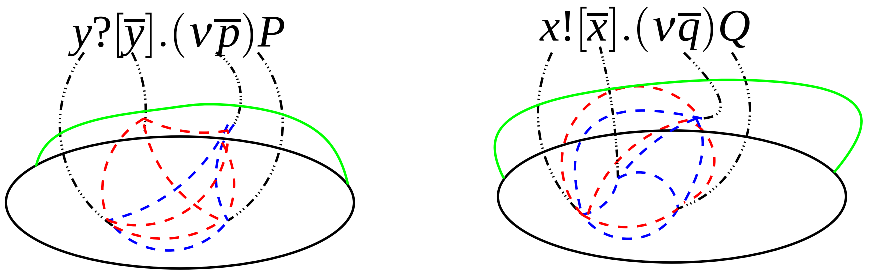
Control flow analyses consists in detecting the origin of all the values that may be associated to each variable of each thread. Thus each variable is dealt with separately, which means that we do not carry out relational information about the values that are associated with distinct variables of the same thread instances. More relational abstraction are designed in Sect. 8.3. We relate each variable of each syntactic component to the set of the labels of the values that they may be


$y?[\overline{y}].(\nu\overline{p})P$
$x![\overline{x}].(\nu\overline{q})Q$
?

(a) Enabling the interaction

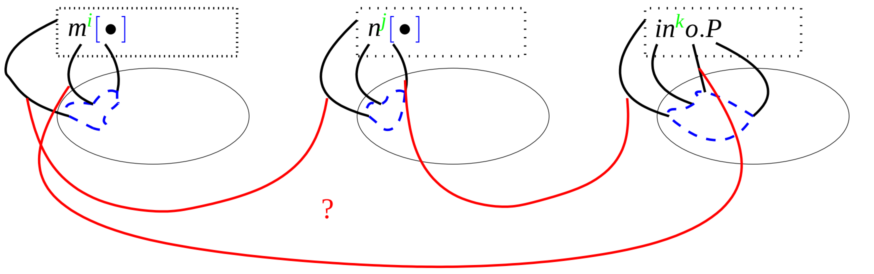

$y?[\overline{y}].(\nu\overline{p})P$
$x![\overline{x}].(\nu\overline{q})Q$

(b) Synchronization

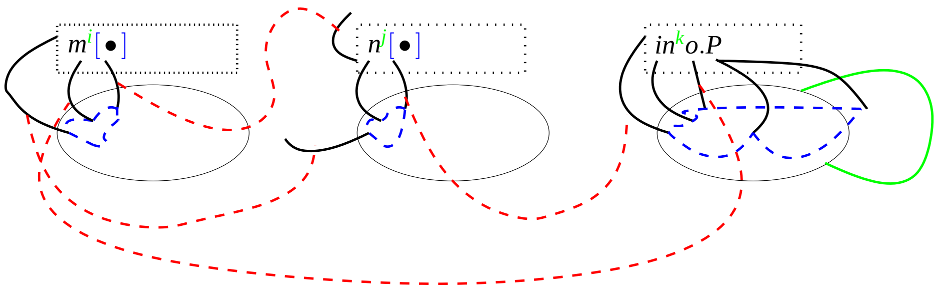

$y?[\overline{y}].(\nu\overline{p})P$
$x![\overline{x}].(\nu\overline{q})Q$

(c) Information closure

Figure 8.2: Abstract communication.



(a) Enabling the interaction



(b) Synchronization

Figure 8.3: Abstract migration.

associated with. However, our framework allows the description of non-uniform properties. This means that we can distinguish each instance of the same thread and each instance of the same value. To achieve this goal, we also compute a comparison between the markers of the threads which declare the values and the markers of the threads that receive these values. Due to the approximation, some of the discovered interactions may be ineffective. But the analyzer detects all the interactions. So, if the analyzer does not detect any interaction between two components, there cannot be any interaction between them.

## 8.2.1 Generic marker abstraction

### 8.2.1.1 Abstract marker and abstract marker pair

We introduce here several generic domains for describing both markers and relations among markers. We suppose that the abstract pre-order $(\mathcal{M}_1^\sharp, \sqsubseteq_1^{\mathcal{M}})$ describes sets of markers: it is related to the concrete domain $\wp(\mathcal{M})$ by a monotonic concretization function $\gamma_1^{\mathcal{M}}$. Furthermore, the abstract pre-order $(\mathcal{M}_2^\sharp, \sqsubseteq_2^{\mathcal{M}})$ describes sets of marker pairs. This domain is used for describing relations between markers. It especially abstracts the relations between thread markers and value markers. The domain $\mathcal{M}_2^\sharp$ is related to the concrete domain $\wp(\mathcal{M}^2)$ by a monotonic concretization function $\gamma_2^{\mathcal{M}}$.

We introduce some primitive to handle these domains: the representations of the empty set $\bot_1^{\mathscr{M}} \in \mathscr{M}_1^\sharp$ and $\bot_2^{\mathscr{M}} \in \mathscr{M}_2^\sharp$; a representation of the set of all the markers $\top_1^{\mathscr{M}}$; a representation of the initial marker $\varepsilon_1^{\mathscr{M}} \in \mathscr{M}_1^\sharp$; abstract unions $\sqcup_1^{\mathscr{M}}$ and $\sqcup_2^{\mathscr{M}}$; abstract intersections $\sqcap_1^{\mathscr{M}}$ and $\sqcap_2^{\mathscr{M}}$; widening operators $\nabla_i^{\mathscr{M}}$ and $\nabla_2^{\mathscr{M}}$. We also introduce the abstract counterpart to the pair constructor $\text{PAIR} \in (\mathscr{M}_1^\sharp)^2 \to \mathscr{M}_2^\sharp$. Then a primitive primitive DIAG removes all the marker pair the two components of which are not the same.

These primitives shall satisfy the following properties:

1. $\gamma_i^{\mathscr{M}}(\bot_i^{\mathscr{M}}) = \emptyset$, for any $i \in \{1;2\}$;

2. $\mathscr{M} \subseteq \gamma_1^{\mathscr{M}}(\top_1^{\mathscr{M}})$;

3. $\varepsilon \in \gamma_1^{\mathscr{M}}(\varepsilon_1^{\mathscr{M}})$;

4. $\forall i \in \{1;2\}, \forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp), \sqcup_i^{\mathscr{M}}(A) \in \mathscr{M}_i^\sharp$ and $\forall a \in A, a \sqsubseteq_i^{\mathscr{M}} \sqcup_i^{\mathscr{M}}(A)$;

5. $\forall i \in \{1;2\}, \forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp), \sqcap_i^{\mathscr{M}}(A) \in \mathscr{M}_i^\sharp$ and $\cap(\{\gamma_i^{\mathscr{M}}(a) \mid a \in A\}) \subseteq \gamma_i^{\mathscr{M}}(\sqcap_i^{\mathscr{M}}(A))$;

6. $\forall i \in \{1;2\}, \nabla_i^{\mathscr{M}} : (\mathscr{M}_i^\sharp)^2 \to \mathscr{M}_i^\sharp$ is a widening operator;

7. $\forall a,b \in \mathscr{M}_1^\sharp, \text{PAIR}(a,b) \in \mathscr{M}_2^\sharp$ and $\gamma_1^{\mathscr{M}}(a) \times \gamma_1^{\mathscr{M}}(b) \subseteq \gamma_2^{\mathscr{M}}(\text{PAIR}(a,b))$;

8. $\forall a \in \mathscr{M}_2^\sharp, \text{DIAG}(a) \in \mathscr{M}_2^\sharp$ and $\gamma_2^{\mathscr{M}}(a) \cap \{(id,id) \mid id \in \mathscr{M}\} \subseteq \gamma_2^{\mathscr{M}}(\text{DIAG}(a))$.

#### 8.2.1.2 Abstract marker function

Since we want to describe an interaction between several threads, we need to take into account the relations between much more markers. For that purpose we introduce a domain that encodes functions the co-domain of which is the set of the markers. For each finite set $X$, the abstract domain $\mathscr{M}_X^\sharp$ is related to the concrete domain $\wp(X \to \mathscr{M})$ by a concretization map $\gamma_X^{\mathscr{M}}$. The domain $\mathscr{M}_X^\sharp$ is not fitted with any structure: it is only use in intermediary step and it is never involved in a post-fixpoint extrapolation. Thus, each domain $\mathscr{M}_X^\sharp$ is related to $\wp(X \to \mathscr{M})$ by a monotonic concretization function $\gamma_X^{\mathscr{M}}$.

For each finite set $X$, we introduce some abstract primitives to handle these domains: the representation of the empty set $\bot_X^{\mathscr{M}}$, the representation of the empty function $\emptyset_{\mathscr{M}}^\sharp$ (that abstracts the function $\emptyset_{\mathscr{M}} \in \emptyset \to \mathscr{M}$ the domain of which is empty), an abstract union $\sqcup_X^{\mathscr{M}}$, an abstract quotient to enforce synchronization conditions[3] and an abstract push operator *push* which is used to compute the abstraction of the set of the new markers when replicating a resource.

1. $\gamma_X^{\mathscr{M}}(\perp_X^{\mathscr{M}}) = \emptyset$;
2. $\gamma_X^{\mathscr{M}}(\emptyset^{\sharp}_{\mathscr{M}}) = \{\emptyset_{\mathscr{M}}\}$;
3. $\forall A \in \wp_{\text{finite}}(\mathscr{M}_X^{\sharp}),\ \sqcup_X^{\mathscr{M}}(A) \in \mathscr{M}_X^{\sharp}$ and $\forall a \in A,\ a \sqsubseteq_X^{\mathscr{M}} \sqcup_X^{\mathscr{M}}(A)$;
4. $\forall q \in X \to Y,\ \forall a \in \mathscr{M}_X^{\sharp},\ \textsc{quotient}(q,a) \in \mathscr{M}_Y^{\sharp}$ and
   $\{f \in Y \to \mathscr{M} \mid (f \circ q) \in \gamma_X^{\mathscr{M}}(a)\} \subseteq \gamma_Y^{\mathscr{M}}(\textsc{quotient}(q,a))$
5. Let $n$ be an integer. Let $(p^k)_{1 \le k \le n}$ be an $n$-tuple of program point labels. We suppose that $\{(I,i) \mid 1 \le i \le n\} \subseteq X$, then $\forall a \in \mathscr{M}_X^{\sharp},\ \underset{(p^k)}{\textsc{push}}(a) \in \mathscr{M}_X^{\sharp}$ and
   $\{f[(I,1) \mapsto N((p^k), f(I,1), \ldots, f(I,k))] \mid f \in \gamma_X^{\mathscr{M}}(a)\} \subseteq \gamma_X^{\mathscr{M}}(\underset{(p^k)}{\textsc{push}}(a))$.

The abstract quotient has two distinct purposes. The first one is re-indexing (when it is applied with an injective function) and the second is merging some variables while synchronizing their values (when it is applied with a non-injective function).

We also introduce an associative abstract join operator $\otimes$ and an abstract projection PROJ that are correct counterparts to the concrete join operator [4] and to the concrete projection operator. These primitives shall satisfy the following properties:

1. $\forall X,Y,\ \forall a \in \mathscr{M}_X^{\sharp},\ b \in \mathscr{M}_Y^{\sharp},\ (a \otimes b) \in \mathscr{M}_{X\cup Y}^{\sharp}$ and
   $\big\{(f \in (X \cup Y \to \mathscr{M}) \mid f_{|X} \in \gamma_X^{\mathscr{M}}(a),\ f_{|Y} \in \gamma_Y^{\mathscr{M}}(b)\big\} \subseteq \gamma_{X\cup Y}^{\mathscr{M}}(a \otimes b)$;
2. $\forall a \in \mathscr{M}_X^{\sharp},\ \forall Y \subseteq X,$
   $\textsc{proj}(Y,a) \in \mathscr{M}_Y^{\sharp}$ and $\{f_{|Y} \mid f \in \gamma_X^{\mathscr{M}}(a)\} \subseteq \gamma_Y^{\mathscr{M}}(\textsc{proj}(Y,a))$;

We assume that the join operator $\otimes$ is associative and commutative. Thus we can define the operator $\bigotimes$ that takes a set of abstract molecules and fold the $\otimes$ operator onto the elements of this set, starting from the $\emptyset^{\sharp}_{\mathscr{M}}$ elements.

[3]The abstract quotient of an abstract element $a \in \mathscr{M}_X^{\sharp}$ by a function $q \in X \to Y$ extracts all the functions $f$ that satisfies both the property $a$ and the assertion $[q(x) = q(y) \implies f(x) = f(y)]$ and then replaces the domain $X$ of $f$ with the co-domain $Y$ of $q$.

[4]The join of two functions sets $A \in \wp(X \to I)$ and $B \in \wp(Y \to I)$ is obtained by merging each function pair $(f,g) \in A \times B$ that $f$ and $g$ coincide on $X \cap Y$

#### 8.2.1.3 Conversion primitive

Last we introduce some primitives to relate the domain $\mathscr{M}_1^\sharp$ to the domain $\mathscr{M}_s^\sharp$ for any singleton $s$ and the domain $\mathscr{M}_2^\sharp$ to the domain $\mathscr{M}_p^\sharp$ for any pair $p$. Let $X$ be a set of elements. Let $x$ and $y$ be two distinct elements in $X$. We introduce the primitives BUILD-SG, EXTRACT-SG, BUILD-PR and EXTRACT-PR as follows:

- $\forall a \in \mathscr{M}_1^\sharp$, $\textsc{build-sg}_x(a) \in \mathscr{M}_{\{x\}}^\sharp$ and
  $\gamma_1^{\mathscr{M}}(a) \subseteq \{f(x) \mid f \in \gamma_{\{x\}}^{\mathscr{M}}(\textsc{build-sg}_x(a)\}$;
- $\forall a \in \mathscr{M}_X^\sharp$, $\textsc{extract-sg}_x(a) \in \mathscr{M}_1^\sharp$ and
  $\{f(x) \mid f \in \gamma_X^{\mathscr{M}}(a)\} \subseteq \gamma_1^{\mathscr{M}}(\textsc{extract-sg}_x(a))$;
- $\forall a \in \mathscr{M}_2^\sharp$, $\textsc{build-pr}_{x,y}(a) \in \mathscr{M}_{\{x;y\}}^\sharp$ and
  $\gamma_2^{\mathscr{M}}(a) \subseteq \{(f(x), f(y)) \mid f \in \gamma_{\{x;y\}}^{\mathscr{M}}(\textsc{build-pr}_{x,y}(a))$;
- $\forall a \in \mathscr{M}_X^\sharp$, $\textsc{extract-pr}_{x,y}(a) \in \mathscr{M}_2^\sharp$ and
  $\{(f(x), f(y)) \mid f \in \gamma_{\{x;y\}}^{\mathscr{M}}(a)\} \subseteq \gamma_2^{\mathscr{M}}(\textsc{extract-pr}_{x,y}(a))$.

### 8.2.2 Atom abstraction

We abstract separately the environment of each thread: for each program point, we approximate the values that may be associated with each variable of the interface of a thread at this program point. We will also capture pair-wise comparisons between the marker of the thread, and the marker of the values of each of its variables.

Thus for each subset $V \subseteq \mathscr{V}$, we introduce the domain $Atom_V^\sharp = \mathscr{M}_1^\sharp \times (V \times \mathscr{L} \to \mathscr{M}_2^\sharp)$. Each pair $(a,b) \in Atom_V^\sharp$ is related to the set $\gamma_V((a,b))$ of the pair $(id, E) \in \mathscr{M} \times (V \to (\mathscr{L} \times \mathscr{M}))$ such that:

1. $id \in \gamma_1^{\mathscr{M}}(a)$;
2. $E(x) = (y, id_y) \implies (id, id_y) \in \gamma_2^{\mathscr{M}}(b(x,y))$.

This way, the first component $a$ describes the thread marker whereas the second component $b$ maps each pair $(x,y) \in \mathscr{V} \times \mathscr{L}$ into an abstraction of the pair of markers $(id, id_x)$, such that the thread may be associated with the marker $id$ while the variable $x$ is associated in its environment with the value $(y, id_y)$.

The structure of these domains is the component-wise extension of the structure of the domains $\mathscr{M}_1^\sharp$ and $\mathscr{M}_2^\sharp$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:

  $$\varepsilon_{\emptyset}^{\sharp} = (\varepsilon_1^{\mathscr{M}}, \emptyset)$$

  where $\emptyset$ denotes the function the domain of which is the empty set.

- the abstract restriction is defined as follows:

  $$\nu^{\sharp}(x, l, (a, b)) = (a, b')$$

  where $b' = \begin{cases} (y,k) \mapsto b(y,k) & \text{if } x \neq y, \\ (x,k) \mapsto \bot_2^{\mathscr{M}} & \text{if } k \neq l, \\ (x,l) \mapsto \text{DIAG}(\text{PAIR}(a,a)) & \text{otherwise.} \end{cases}$

  This way, information about variables $y \neq x$ remains unchanged. We also encode that the label of the value of $x$ is necessarily $l$. We use the primitive DIAG to encode the fact that the marker of the value of $x$ is the marker of the thread.

- the abstract garbage collection is defined as follows:

  $$\text{GC}^{\sharp}(X, (a, b)) = (a, b_{|(V \cap X) \times \mathscr{L}}),$$

  by removing any information about garbage collected variables.

**Theorem 8.2.1.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

### 8.2.3 Molecule abstraction

The main difficulty is to synthesize comparisons between markers throughout computation steps. We use the marker of the thread instance as a pivot to synthesize the comparison between the markers of the values. Furthermore, we use synchronization conditions on the values to establish a comparison between the markers of all the involved thread instances. Our main strategy is easy: first we gather all the information we have about involved markers (this means we will abstract sets of marker functions). Then, synchronization conditions give equality relations. If these equality relations are satisfiable, the abstract computation step is enabled and we compute, for each new thread instance, the comparison between the marker of the new instance and the markers of the values.

#### 8.2.3.1 Domain intuition

To achieve this goal, we will store abstract relations about the markers which are involved in the synchronization constraints of the computation step. In order to get a better accuracy, we partition this abstraction according to the label of the values that are associated to the synchronized variables.

Thus an abstract molecule is described by five components:

- The first component is a tuple of abstract atoms: it gives a description of each interacting thread separately.

- The second component is a set of variables. This set contains all the variables that are synchronized during the computation step.

- The third component approximates some equality relations about the variables that are constrained by the synchronization. It is given by an equivalence relationship among the variables that are involved in the synchronization.

- The fourth component approximates some disequality relations about the variables that are constrained by the synchronization: some equivalence classes are related with an edge that encodes the fact that the variables that they contain are associated with distinct values.

- The fifth component describes the binding of constrained variables. This description is partitioned according to the syntactic labels of the values: it is given by a partial function that maps any function that associates equivalence classes into a syntactic label to an abstraction of the markers that are associated with the synchronized variables whenever they are associated with these labels.

#### 8.2.3.2 Domain definition

More formally, the domain $\mathit{Molecule}^{\sharp}_{(V_i)_{i\in[\![1;n]\!]}}$ is the set of all the 5-tuples $(f,S,C,E,r)$ where:

- the element $f \in [\![1;n]\!] \rightarrow \mathit{Atom}^{\sharp}_{V_i}$ is a map;

- the set $S \subseteq \{(a,k) \mid 1 \leq k \leq n,\ a \in V_k \cup \{I\}\}$ is the set of the synchronized variables (the variable $(I,k)$ denotes the identity of the $k$-th interacting thread while the variable $(a,k)$ with $a \in V_k$ denotes the variable $a$ of the $k$-th interacting component;

- the set $P \in \wp(\wp(S))$ is a partition of $V$. It means that two variables of the same partition class in $P$ are necessarily associated with the same value;
- the set $E \in P \times P$ denotes disequality relations among the elements of the partition classes: it means that the variables of two related classes are necessarily associated with distinct values;
- the map $r \in (P \to \mathscr{L}) \to \mathscr{M}_P^\sharp$ describes information about the values associated with the synchronized variables. This information is partitioned according to the labels of the values that are associated with the variables of each partition class. For each valuation function $t \in P \to \mathscr{L}$, the abstract element $r(t) \in \mathscr{M}_P^\sharp$ describes relational information about the markers of the threads and about the markers of the synchronized variable values whenever in each class $C$, the variables in the class $C$ are associated with a value tagged with the label $t(C)$.

The concretization of an element $(f,S,P,E,r) \in Molecule^\sharp_{(V_i)_{i\in[\![1;n]\!]}}$ is defined as the set of the $n$-tuples $(p_i,id_i,E_i)_{i\in[\![1;n]\!]}$ of threads that satisfy:

1. For any $i \in [\![1;n]\!]$, we have $(id_i,E_i) \in \gamma_{V_i}(f(i))$.
2. We introduce the map $\sigma$ that interprets the variables as follows:
$$\sigma : \begin{cases} (I,k) \to (p_k,id_k) \\ (a,k) \to E_k(a). \end{cases}$$
Then
   - for any $C \in P$, we have:
$$\forall x,y \in C,\ \sigma(x) = \sigma(y);$$
   - for any $(C_1,C_2) \in E$, we have:
$$\forall x \in C_1, y \in C_2,\ \sigma(x) \neq \sigma(y).$$
3. Let us introduce the function $t \in P \to \mathscr{L}$ that associates each equivalence class $C \in P$ with the label of the value which is associated with any variable of the class $C$. This is defined[5] by $t(C) = l$ when there exists $X \in C$ and a marker $id$ such that $\sigma(X) = (l,id)$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P \to \mathscr{M}$ that associates each equivalence class $C \in P$ with the marker of the value which is associated with any variable of the class $C$. The function $f_{\mathscr{M}}$ is defined by $f_{\mathscr{M}}(C) = id$ when there exists $X \in C$ and a label $l \in \mathscr{L}$ such that $\sigma(X) = (l,id)$. Then we have $f_{\mathscr{M}} \in \gamma_P^{\mathscr{M}}(r(t))$.

[5]The definition is well-formed due to the definition of the equivalence relation.

#### 8.2.3.3 Abstract primitives

We now define the primitives for our domain:

- *abstract injection:* The abstract injection just consists in letting the atom abstraction unchanged, there is no synchronized variables yet, the graph is empty, and no information are stored about the values that are associated with the elements of equivalence classes. Thus we define:

$$\text{INJ}^\sharp(a) = ([1 \mapsto a], \emptyset, \emptyset, \emptyset, [\emptyset_{\mathscr{L}} \to \emptyset^\sharp_{\mathscr{M}}]).$$

- *abstract product:* When gathering some abstract molecules, we just rename the index that occur in the right argument. Let $(f,S,P,E,r) \in Molecule^\sharp_{(V_i)_{i\in[\![1;n]\!]}}$ and $(f',S',P',E',r') \in Molecule^\sharp_{(V'_i)_{i\in[\![1;n']\!]}}$. We set $n'' = n+n'$ and $(V''_i)_{i\in[\![1;n'']\!]}$ that is defined as $V''_i = V_i$ for any $i \in [\![1;n]\!]$, and as $V''_{i+n} = V'_i$ for any $i \in [\![1;n']\!]$. The variables that occur in $(f',S',P',E',r'')$ must be re-indexed before gathering the two abstract elements. For that purpose, we introduce several re-indexing functions as follows:

  1. for each variable $(x,k) \in S'$, we define the variable $\sigma_v(x,k)$ as $(x,k+n)$;
  2. for any subset $A \subseteq S'$, we define the set $\sigma_{\wp}(A)$ as $\{\sigma_v(x) \mid x \in A\}$;
  3. for any relation $R \in \wp(S') \times \wp(S')$, we define the relation $\sigma_{\mathscr{R}}(R)$ as $\{(\sigma_{\wp}(C_1), \sigma_{\wp}(C_2)) \mid (C_1,C_2) \in R\}$.

  Then we define the abstract concatenation $(f,S,P,E,r) \bullet (f',S',P',E',r')$ as $(f'',S'',P'',E'',r'') \in Molecule^\sharp_{(V''_i)}$ where:

  - $f'' : \begin{cases} i \mapsto f(i) & \text{when } i \le n \\ i \mapsto f'(i-n) & \text{when } i > n; \end{cases}$
  - $S'' = S \cup \sigma_{\wp}(S')$:
  - $P'' = P \cup \{\sigma_{\wp}(C) \mid C \in P'\}$;
  - $E'' = E \cup \sigma_{\mathscr{R}}(E')$;
  - $r''(t) = r(t_{|P}) \otimes \text{QUOTIENT}\left(\sigma_v, r'\left([C \in P' \mapsto t(\sigma_{\wp}(C))]\right)\right)$, where the primitive QUOTIENT is used only for re-indexing purposes.

- *abstract projections:* Let $(f,S,P,E,r) \in Molecule^\sharp_{(V_i)}$ be an abstract elements. For any $X \in S$, we denote by $C(X)$ the element in $P$ such that $X \in C(X)$. Before performing the abstract projection, we use the information collected about the variables in $S$ to refine the atom abstraction $f(k)$.

Let us denote by $(a_0,b_0)$ the abstract atom $f(k)$. We define the abstract projection $\textsc{proj}^\sharp(k,(f,S,P,E,r))$ as the abstract atom $(a,b)$ where the abstract thread marker $a \in \mathscr{M}_1^\sharp$ is given by:

$$\begin{cases} \sqcap_1^{\mathscr{M}}(\{a_0;\sqcup_1^{\mathscr{M}}(\{\textsc{extract-sg}_{C((I,k))}(r(\rho)) \mid \rho \in P \to \mathscr{L}\})\}) & \text{if } (I,k) \in S \\ a_0 & \text{otherwise,} \end{cases}$$

and for any $(x,y) \in \mathscr{V} \times \mathscr{L}$, the abstract marker pair $b(x,y) \in \mathscr{M}_2^\sharp$ is given by $\sqcap_2^{\mathscr{M}}\{b_0(x,y);A\}$ where the marker pair abstraction $A$ denotes the new constraints that may be collected by merging all the cases of the partition. The marker pair abstraction $A$ is defined as follows:

1. in the case when $(I,k) \in S$ and $(x,k) \in S$:

$$A \triangleq \sqcup_2^{\mathscr{M}}\left(\left\{\textsc{extract-pr}_{C((I,k)),C((x,k))}(r(\rho)) \,\middle|\, \begin{array}{l}\rho \in P \to \mathscr{L},\\ \rho(C((x,k))) = y\end{array}\right\}\right);$$

2. in the case when $(I,k) \in S$ and $(x,k) \notin S$:

$$A \triangleq \textsc{pair}(a,\top_1^{\mathscr{M}});$$

3. in the case when $(I,k) \notin S$ and $(x,k) \in S$:

$$A \triangleq \textsc{pair}\left(a,\sqcup_1^{\mathscr{M}}\left\{\textsc{extract-sg}_{C((x,k))}(r(\rho)) \,\middle|\, \begin{array}{l}\rho \in P \to \mathscr{L},\\ \rho(C((x,k))) = y\end{array}\right\}\right);$$

4. in the case when $(I,k) \notin S$ and $(x,k) \notin S$:

$$A \triangleq \textsc{pair}(\top_1^{\mathscr{M}},\top_1^{\mathscr{M}}).$$

- *abstract extension:* Let $(V_i)$ be a family of $n$ interfaces, let $X$ be a subset of $\mathscr{V} \times [\![1;n]\!]$, and let $(f,S,P,E,r)$ be an abstract element in $Molecule^\sharp_{(V_i)}$. Abstract extension consists in associating new variables with no information in their abstract atoms and in removing any constraints about the variable in $X$ in the abstraction of the thread relations. For each $i \in [\![1;n]\!]$, we define the set $W_i \subseteq \mathscr{V}$ of variables as $V_i \cup \{x \mid (x,i) \in X\}$. We define the abstract extension, $\textsc{new}^\sharp_\top(X,A) \in Molecule^\sharp_{(W_i)}$, of $A$ by $X$, as $(f',S',P',E',r')$ where:

  1. $f'(i) = \left(id^\sharp, \begin{cases}(x,y) \mapsto \textsc{pair}(id^\sharp,\top_1^{\mathscr{M}}) & \text{if } (x,i) \in X \\ (x,y) \mapsto g^\sharp(x,y) & \text{otherwise}\end{cases}\right)$

     where $(id^\sharp,g^\sharp) = f(i)$;

2. $S' = S \setminus X$;
3. $P' = \{C_1 \setminus X \mid C_1 \in P,\ C_1 \not\subseteq X\}$;
4. $E' = \{(C_1 \setminus X, C_2 \setminus X) \mid (C_1, C_2) \in E,\ C_1 \not\subseteq X,\ C_2 \not\subseteq X\}$;
5. $r'(\rho) = \sqcup^{\mathscr{M}}_{P'} \{\psi(r(\theta)) \mid \forall C \in P,\ C \not\subseteq X \implies \theta(C) = \rho(C \setminus X)\}$, where $\psi(a)$ is defined as:

$$\text{QUOTIENT}([C \mapsto C \setminus X], \text{PROJ}(\{C \in P \mid C \not\subseteq X\}, a)).$$

- *abstract synchronization:* Let $n \in \mathbb{N}$ be an integer. Let $(V_i)$ be a family of $n$ interfaces. Let $(f, S, P, E, r)$ be an abstract in $Molecule^{\sharp}_{(V_i)_{1 \leq i \leq n}}$, let $(p_i) \in \mathscr{L}_{\text{p}}^n$ be a tuple of program point labels and $A$ be a set of constraints of the form $(x,k) \diamond (y,l)$ where $k, l \in [\![1;n]\!]$, $x \in V_k \cup \{I\}$ and $y \in V_l \cup \{I\}$, the abstract synchronization $\text{SYNC}^{\sharp}(A, (p_i), (f, S, P, E, r))$ of $(f, D, P, E, r)$ according to the set of constraints $A$ is computed in several steps:

  1. First we compute the set of the variables that are constrained. We introduce the set $V_s$ of variables that occurs in the synchronization constraints: we set $V_s = \{a \in \mathscr{V} \times \mathbb{N} \mid \exists b, \diamond,\ (a \diamond b) \in A \textit{ or } (b \diamond a) \in A\}$. Constraining a variable also provide information about the identity of the thread. We define the set of the constrained variables $V_n$ as the set $V_s \cup \{(I,k) \mid \exists a \in \mathscr{V},\ (a,k) \in V_s\}$.
  2. Then we update the set of the constrained variables: We define $S_1 = S \cup V_n$. At first, we associate no information with these variables. So we set $P_1 = P \cup \{\{x\} \mid x \in S_1 \setminus S\}$ and $E_1 = E$.
  3. We now take into account equality synchronization constraints. We update equivalence classes. We define a binary equivalence $\sim$ over $P_1$ as the smaller (as a subset of $(P_1)^2$) equivalence such that for any classes $C_1, C_2 \in P_1$ such that there exist two variables $a \in C_1$ and $b \in C_2$ that satisfy $(a = b) \in A$, we have $C_1 \sim C_2$. Then we merge the equivalence classes: we define $P_2$ as:

$$\left\{ \bigcup \{Y \in P_1 \mid X \sim Y\} \mid X \in P_1 \right\}.$$

  4. Disequality edges collect previous disequality relations and the new ones. We define $E_2$ as $old \cup new$ where:

$$\begin{cases} old = \{(X,Y) \in (P_2)^2 \mid \exists (Z,T) \in E_1, Z \subseteq X \text{ and } T \subseteq Y\}, \\ new = \{(X,Y) \in (P_2)^2 \mid \exists (x,y) \in X \times Y,\ a \neq b \in A\}. \end{cases}$$

5. The fourth step update the definition of $r$. First we introduce a quotient function $q$ that maps any class $C \in P$ into the unique class $q(C) \in P_2$ such that $C \subseteq q(C)$. Moreover, for each element $x \in V_1$, we denote by $[x]$ the unique class $[x] \in P_2$ such that $x \in [x]$. Then we define $r_1$ as follows:

$$r_1(\rho) = \begin{cases} \bigotimes(A_1(\rho) \cup A_2(\rho) \cup A_3(\rho)), & \text{if } \rho([(I,k)]) = p_k,\ \forall k \in [\![1;n]\!], \\ \bot^{\mathscr{M}}_{P_2}, & \text{otherwise,} \end{cases}$$

where:

- $A_1(\rho) = \{\textsc{quotient}(q, r([A \in S \mapsto \rho(q(A))]))\}$,
- $A_2(\rho) = \{\textsc{build-pr}_{[(I,k)],[(a,k)]}(snd(f(k))(a, \rho([(a,k)]))) \mid (a,k) \in V_s\}$,
- $A_3(\rho) = \{\textsc{build-sg}_{[(I,k)]}(fst(f(k))) \mid \exists a \in \mathscr{V},\ (a,k) \in V_s\}$,

Intuitively, the computation of $r_1(\rho)$ consists in taking the join of three sets of constraints: the first one $A_1(\rho)$ contains the constraints that are given by quotienting the old ones; the second one $A_2(\rho)$ collects the constraints about marker pair from the abstraction of atoms; the third one $A_3(\rho)$ collects the constraints about marker identities from the abstraction of atoms. Moreover, if $\rho$ does not map the identity of each thread to its program point label, the $r_1(\rho)$ is not satisfiable.

Either in the case when there exists $X \in P_2$ such that $(X,X) \in E_2$ or in the case when we have $r_1(\rho) = \bot^{\mathscr{M}}_{P_2}$ for any $\rho \in P_2 \to \mathscr{L}$, the synchronization is not possible in the concrete level. In such cases, we set $\textsc{sync}^\sharp(A,(p_i),(f,S,P,E,r)) = \bot^{\mathscr{M}}_{(V_i)}$, otherwise we set $\textsc{sync}^\sharp(A,(p_i),(f,S,P,E,r)) = (f,S_1,P_2,E_2,r_1)$. We notice that the first component is left unchanged, reduction will be performed later when doing projections.

- *abstract marker allocation:* To simulate marker allocation in the abstract, we need to update the relational information between the values that are associated with each variable of the replicated threads and the marker of these threads. This is performed in several steps.

  Let $n \in \mathbb{N}$ be an integer. Let $(V_i)$ be a family of $n$ interfaces. Let $(p^k)$ be an $n$-tuple of program points. Let $(f,S,P,E,r)$ be an abstract molecule in $\mathit{Molecule}^\sharp_{(V_i)}$.

  1. Since the new thread marker is built from all markers of the threads that are interacting, we add any variable of the form $(I,k)$ to the set

of the constrained variables: we set $S_1 = S \cup \{(I,k) \mid 1 \leq k \leq n\}$ and $P_1 = P \cup \{\{(I,k) \mid 1 \leq k \leq n, (I,k) \notin S\}\}$. Then we update the map $r$ by joining previous constraints with marker abstraction that may be collected from the abstract atoms: for any $\rho \in S_1 \to \mathscr{L}$, we set $r_1(\rho) = \bigotimes\{r(\rho_{|S})\} \cup \{[(I,k) \mapsto fst(f(k))] \mid 1 \leq k \leq n,\ (I,k) \notin S\}$ if for any $k$ such that $1 \leq k \leq n$ and $(I,k) \notin S$, we have $\rho(\{(I,k)\}) = p^k$, and we set $r_1(\rho) = \bot^{\mathscr{M}}_{P_1}$ otherwise.

2. The operator PUSH requires that its argument is an abstraction of marker functions the domain of which contains the variables $\{(I,k) \mid 1 \leq k \leq n\}$. But, for any valuation $\rho \in P_1 \to \mathscr{L}$, the element $r_1(\rho)$ abstracts marker functions the domain of which is class of variables. Thus, we extend function domains by duplicating information about each class that contains a variable of the form $(I,k)$. For any $k$ such that $1 \leq k \leq n$, we define $\phi_k$ as $\phi_k(X) = (I,k)$ when $(I,k) \in X$, and $\phi_k(X) = X$ otherwise. Then, for any element $a \in P_1$, we define $\psi(a) \in \mathscr{M}^{\sharp}_{P_1 \cup \{(I,k) \mid 1 \leq k \leq n\}}$ as $\bigotimes\{\text{QUOTIENT}(\phi_k, a) \mid 1 \leq k \leq n\}$.

3. Thus the molecule $(f, S_1, P_1, E, r_1)$ contains all the relational information we need about the markers of the threads that are interacting. Then we compute the abstraction of the marker that is allocated to the new replicated threads: each valuation $\rho \in P_1 \to \mathscr{L}$ gives a thread marker $id^{\sharp}(\rho)$ that is defined as:

$$\text{EXTRACT-SG}_{(I,1)}(\underset{(p^k)}{\text{PUSH}}(\psi(r_1(\rho)))).$$

Then we consider all the cases for the partition and define $\overline{id}^{\sharp}$ as:

$$\sqcup^{\mathscr{M}}_1 \{id^{\sharp}(\rho) \mid \rho \in P_1 \to \mathscr{L}\}.$$

4. The next step consists in computing for each variable $v$ in the interface $I(p^1)$ and each label $l \in \mathscr{L}$, the abstraction of the marker pairs $(id, id_v)$ such that $v$ may be associated with the value $(l, id_v)$ in a thread at program point $p^1$ with a marker $id$. Only the first thread changed. We fix $(v,l) \in (I(p^1) \times \mathscr{L})$. We consider all potential valuation $\rho \in P_1 \to \mathscr{L}$. For any $\rho \in P_1 \to \mathscr{L}$, we define $id^{\sharp}_2(v,l,\rho)$ as $\bot^{\mathscr{M}}_2$ in the case when $\rho(v) \neq l$ or as $\text{EXTRACT-PR}_{(I,1),(v,1)}(\underset{(p^k)}{\text{PUSH}}(\psi(r_1(\rho))))$ otherwise.

Then we consider all the cases for the partition and define $\overline{id}^{\sharp}_2(v,l)$ as:

$$\sqcup^{\mathscr{M}}_2 \{id^{\sharp}_2(v,l,\rho) \mid \rho \in P_1 \to \mathscr{L}\}.$$

Thus we update the definition of $f$. We define $f_1$ as follows:

$$f_1(k) \triangleq \begin{cases} (\overline{id}^{\sharp}, [(v,l) \mapsto \overline{id}_2^{\sharp}(v,l)]) & \textit{if } k = 1 \\ f(k) & \textit{otherwise}. \end{cases}$$

5. The last step updates the information about the variable $(I,1)$: we remove the variable $(I,1)$ from any partition class and enforce the fact that its value is fresh. Then we compute relational information. We set:

$$\begin{cases} P_2 = (\{C \setminus \{(I,1)\} \mid C \in P_1\} \setminus \{\emptyset\}) \cup \{\{(I,1)\}\}, \\ E_2 = \quad (\{(C_1 \setminus \{(I,1)\}, C_2 \setminus \{(I,1)\}) \mid (C_1,C_2) \in E_1\} \cap (P_2)^2) \\ \qquad \cup (\{\{(I,1)\}\} \times (P_2 \setminus \{\{(I,1)\}\})), \\ r_2(\rho) = \text{QUOTIENT}(g, \text{GC}^{\sharp}(Y, \underset{(p^k)}{\text{PUSH}}(\psi(r_1(\rho))))). \end{cases}$$

where $g((I,1)) = \{(I,1)\}$ and $g(X) = X$ otherwise,
and $Y = (P_2 \setminus \{\{(I,1)\}\}) \cup \{(I,1)\}$.

Thus we define $\text{FETCH}^{\sharp}((p^k),(f,S,P,E,r))$ as $(f_1,S_1,P_2,E_2,r_2)$.

**Theorem 8.2.2.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

## 8.2.4 Combining marker abstractions

In this section, we show how we may combine marker abstractions and how we may refine marker abstractions.

### 8.2.4.1 Cartesian product

**Proposition 8.2.3 (marker cartesian product).** *Let* $\left(\mathscr{M}_1^{\sharp}, \mathscr{M}_2^{\sharp}, (\mathscr{M}_X^{\sharp})_X\right)$ *and* $\left(\overline{\mathscr{M}}_1^{\sharp}, \overline{\mathscr{M}}_2^{\sharp}, (\overline{\mathscr{M}}_X^{\sharp})\right)$ *be two marker abstractions and their respective abstract primitives. The tuple* $\left(\mathscr{M}_1^{\sharp} \times \overline{\mathscr{M}}_1^{\sharp}, \mathscr{M}_2^{\sharp} \times \overline{\mathscr{M}}_2^{\sharp}, (\mathscr{M}_X^{\sharp} \times \overline{\mathscr{M}}_X^{\sharp})_X\right)$ *is also a marker abstraction, where each abstract primitive is defined pairwise and each concretization function is the intersection of the concretization functions. This marker abstraction is called the Cartesian product of the marker abstractions* $\left(\mathscr{M}_1^{\sharp}, \mathscr{M}_2^{\sharp}, (\mathscr{M}_X^{\sharp})_X\right)$ *and* $\left(\overline{\mathscr{M}}_1^{\sharp}, \overline{\mathscr{M}}_2^{\sharp}, (\overline{\mathscr{M}}_X^{\sharp})_X\right)$.

#### 8.2.4.2 Reduced domain

**Proposition 8.2.4 (marker reduced abstration).** *Let $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ be a marker abstraction and its abstract primitives. Let $(\rho_1, \rho_2, (\rho)_X)$ be a collection of reduction operators such that:*

- $\forall a \in \mathscr{M}_1^\sharp$, $\gamma_1^{\mathscr{M}}(a) \subseteq \gamma_1^{\mathscr{M}}(\rho_1(a))$;
- $\forall a \in \mathscr{M}_2^\sharp$, $\gamma_2^{\mathscr{M}}(a) \subseteq \gamma_2^{\mathscr{M}}(\rho_2(a))$;
- *for any finite set $X$,* $\forall a \in \mathscr{M}_X^\sharp$, $\gamma_X^{\mathscr{M}}(a) \subseteq \gamma_X^{\mathscr{M}}(\rho_X(a))$;

*Then the tuple $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ fitted with the following primitives:*

1. *for any $i \in \{1;2\}$:*
   (a) $\overline{\bot}_i^{\mathscr{M}} = \bot_i^{\mathscr{M}}$;
   (b) $\overline{\top}_i^{\mathscr{M}} = \rho_i(\top_i^{\mathscr{M}})$;
   (c) $\overline{\varepsilon}_1^{\mathscr{M}} = \rho_1(\varepsilon_1^{\mathscr{M}})$;
   (d) $\forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp)$, $\overline{\sqcup}_i^{\mathscr{M}} A = \rho_i(\sqcup_i^{\mathscr{M}}(\{\rho_i(a) \mid a \in A\})$;
   (e) $\forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp)$, $\overline{\sqcap}_i^{\mathscr{M}} A = \rho_i(\sqcap_i^{\mathscr{M}}(\{\rho_i(a) \mid a \in A\})$;
   (f) $\forall a, b \in \mathscr{M}_i^\sharp$, $(a \overline{\nabla}_i^{\mathscr{M}} b) = (a \nabla_i^{\mathscr{M}} \rho_i(b))$;
   (g) $\forall a, b \in \mathscr{M}_1^\sharp$, $\overline{\text{PAIR}}(a,b) = \rho_2(\text{PAIR}(\rho_1(a), \rho_1(b)))$;
   (h) $\forall a \in \mathscr{M}_2^\sharp$, $\overline{\text{DIAG}}(a)) = \rho_2(\text{DIAG}(\rho_2(a)))$;
2. *for any finite set $X$ and $Y$:*
   (a) $\overline{\emptyset^\sharp_{\mathscr{M}}} = \emptyset^\sharp_{\mathscr{M}}$;
   (b) $\overline{\bot}_X^{\mathscr{M}} = \bot_X^{\mathscr{M}}$;
   (c) $\forall A \in \wp_{\text{finite}}(\mathscr{M}_X^\sharp)$,
   $\overline{\sqcup}_X^{\mathscr{M}} A = \rho_X(\sqcup_X^{\mathscr{M}}(\{\rho_X(a) \mid a \in A\})$;
   (d) $\forall q \in X \to Y$, $\forall a \in \mathscr{M}_X^\sharp$,
   $\overline{\text{QUOTIENT}}(q,a) = \rho_Y(\text{QUOTIENT}(q, \rho_X(a)))$.
   (e) *let $n \geq 2$ be an integer such that $\{(I,i) \mid 1 \leq i \leq n\} \subseteq X$, for any tuple $(p^i)_{1 \leq i \leq n}$ of program point labels,* $\forall a \in \mathscr{M}_X^\sharp$, $\underset{(p^k)}{\overline{\text{PUSH}}}(a) = \rho_X(\underset{(p^k)}{\text{PUSH}}(\rho_X(a)))$;

(f) $\forall a \in \mathscr{M}_X^\sharp,\ b \in \mathscr{M}_Y^\sharp,\ (a \overline{\otimes} b) = \rho_{X\cup Y}(\rho_X(a) \otimes \rho_Y(a))$;

(g) $\forall a \in \mathscr{M}_X^\sharp,\ \overline{\text{PROJ}}(X \cap Y, a) = \rho_{X\cap Y}(\text{PROJ}(X \cap Y, \rho_X(a)))$;

3. *let $X$ be a set of elements, let $x$ and $y$ be two distinct elements in $X$,*

   (a) $\forall a \in \mathscr{M}_1^\sharp,\ \overline{\text{BUILD-SG}}_x(a) = \rho_{\{x\}}(\text{BUILD-SG}_x(\rho_1(a)))$;

   (b) $\forall a \in \mathscr{M}_X^\sharp,\ \overline{\text{EXTRACT-SG}}_x(a) = \rho_1(\text{EXTRACT-SG}_x(\rho_X(a)))$;

   (c) $\forall a \in \mathscr{M}_2^\sharp,\ \overline{\text{BUILD-PR}}_{x,y}(a) = \rho_{\{x;y\}}(\text{BUILD-PR}_{x,y}(\rho_2(a)))$;

   (d) $\forall a \in \mathscr{M}_X^\sharp,\ \overline{\text{EXTRACT-PR}}_{x,y}(a) = \rho_2(\text{EXTRACT-PR}_{x,y}(\rho_X(a)))$;

*is also a marker abstraction. We call it the reduction of the marker abstraction $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ by the reduction operators $(\rho_1, \rho_2, (\rho)_X)$.*

### 8.2.5 Three control flow analyses

Various domains can be used to instantiate the parametric domains $\mathscr{M}_1^\sharp$, $\mathscr{M}_2^\sharp$ and $\mathscr{M}_X^\sharp$ depending on the trade-off between complexity and accuracy. We propose three particular instantiations. The first one abstracts away the information about markers. The result is a uniform control flow analysis. The second one keeps only the equality relations among markers which gives an analysis which may express the same kind of properties than group creation type system [16, 15]. The third one allows for algebraic comparisons of markers which is, to the best of our knowledge, beyond the scope of the analyses previously presented in the literature.

#### 8.2.5.1 Uniform control flow analysis

Uniform control flow analysis consists in detecting the potential interactions between syntactic components, without keeping any information about markers. For each variable, it will capture an upper-approximation of the set of the labels of the values that may be associated with this variable. We introduce the total order $(T, \sqsubseteq_T) = \{\bot; \top\}$ with $\bot \sqsubseteq_T \top$. For any set $X$, the domain $T$ is related to the domain $\wp(X)$ by the monotonic concretization function $\gamma_X^T$ that associates $\bot$ with the empty set $\emptyset$ and $\top$ with $X$. We also introduce an abstract union $\sqcup^T \in \wp(T) \to T$ and an abstract intersection $\sqcap^T \in \wp(T) \to T$ as follows:

1. $\forall A \in \wp(T),\ \sqcup^T(A) = \begin{cases} \bot & \text{if } A \subseteq \{\bot\} \\ \top & \text{otherwise;} \end{cases}$

2. $\forall A \in \wp(T),\ \sqcap^T(A) = \begin{cases} \bot & \text{if } \bot \in A \\ \top & \text{otherwise.} \end{cases}$

We instantiate the domains $\mathscr{M}_1^\sharp$, $\mathscr{M}_2^\sharp$ and $\mathscr{M}_X^\sharp$ for each finite set $X$ with the domain $T$. We take $\gamma_1^{\mathscr{M}} = \gamma_{\mathscr{M}}^T$, $\gamma_2^{\mathscr{M}} = \gamma_{\mathscr{M}^2}^T$ and $\gamma_X^{\mathscr{M}} = \gamma_{X\to\mathscr{M}}^T$.

Then the abstract primitives are defined as follows:

1. $\bot_i^{\mathscr{M}} = \bot$, for any $i \in \{1;2\}$; $\bot_X^{\mathscr{M}} = \bot$, for any finite set $X$;
2. $\emptyset_{\mathscr{M}}^\sharp = \top$;
3. $\top_1^{\mathscr{M}} = \top$;
4. $\varepsilon_1^{\mathscr{M}} = \top$;
5. $\forall i \in \{1;2\}$, $\sqcup_i^{\mathscr{M}} = \sqcup^T$ and $\sqcap_i^{\mathscr{M}} = \sqcap^T$; moreover, $\sqcup_X^{\mathscr{M}} = \sqcup^T$ for any finite set $X$;
6. since the domain $T$ is height bounded, we define $a\nabla_i^{\mathscr{M}} b = \sqcup_i^{\mathscr{M}}\{a;b\}$, for any $i \in \{1;2\}$;
7. $\text{PAIR} = [(a,b) \mapsto \sqcap^T\{a;b\}]$;
8. $\text{DIAG} = [a \mapsto a]$;
9. $\text{QUOTIENT} = [(q,a) \to a]$;
10. $\underset{(p^k)}{\text{PUSH}}(a) = a$;
11. $a \otimes b = \sqcap^T\{a;b\}$;[6]
12. $\text{PROJ}(Y,a) = a$;
13. $\text{BUILD-SG}_x(a) = a$, $\text{EXTRACT-SG}_x(a) = a$, $\text{BUILD-PR}_{x,y}(a) = a$ and $\text{EXTRACT-PR}_{x,y}(a) = a$.

The analysis that we get is always at least as precise as the following classical 0-CFA:

- the analysis proposed in [11, 9] for the $\pi$-calculus ;
- the analysis proposed in [59] for the *ambient* calculus ;
- the analysis proposed in [61] for the *spi*-calculus ;
- the analysis proposed in [60] for the BIO-*ambients*.

[6]For the sake of simplicity, we ignore the case when $a$ or $b$ is the abstraction of the empty function.

These analyses computes the least element of a Moore family defined as the solution set of a constraint system. Since our abstract union is exact and because we do not use widening operators, the result of our abstract semantics is the least fixpoint of the abstract endomorphism induced by $C_{0_{env}}$ and $\rightarrow_{env}$. This least fixpoint is also the least element of a Moore family defined as the solution set of a constraint system. Then, comparing the constraints involved in both analyses, it turns out that our constraint system is implied by the 0-CFA system. So, any solution of the 0-CFA is also a solution of our system and the least solution of ours is more precise than the one obtained by using the 0-CFA. Roughly speaking, 0-CFA does not take into account action sequentiality: the constructed system only depends on the syntactic partial interaction set. Furthermore, it may not infer distinct abstractions for two distinct occurrences of the same channel name.

Moreover the analysis described in [60] is refined by an analysis of the threads that may not occur simultaneously. Such refinements are also possible in our framework. Mutual exclusions are dealt in the next chapter.

#### 8.2.5.2 Confinement

We now focus on the equality relations between markers. We use this analysis in order to prove that a value may only be passed to the thread instances with the same marker. It is especially useful in case of recursion or in case of replication to ensure that the values that are declared by some instances may not be passed to others instances of the same thread.

For that purpose, we use a graph-based domain to represent equality relations between markers. Each vertex describes a component; a path between two vertice expresses the fact that the two related components are always equal. Then we lift that domain with an extra element in order to represent a non-satisfiable property. Let $X$ be a finite set. We define the set $\mathscr{G}_X = (\wp(X^2))$ of all the graphs $\curvearrowright$ the set of the vertice of which is $X$. The reflexive, symmetric, and transitive closure of a graph $(\curvearrowright)$ is denoted by $(\curvearrowright^*)$. The pre-order $\sqsubseteq_X^{\mathscr{G}}$, the concretization function $\gamma_X^{\mathscr{G}}$ and some abstract primitives are defined on $\mathscr{G}_X$ as follows:

- $\forall \curvearrowright_1, \curvearrowright_2 \in \wp(X^2), \curvearrowright_1 \sqsubseteq_X^{\mathscr{G}} \curvearrowright_2 \iff \curvearrowright_2 \subseteq \curvearrowright_1,$

- $\forall \curvearrowright \in \wp(X^2),\ \gamma_X^{\mathscr{G}}(\curvearrowright) = \{f \in X \rightarrow \mathscr{M} \mid x \curvearrowright y \implies f(x) = f(y)\};$

- $\forall A \in \wp(X^2) \setminus \emptyset$, we define $\sqcup_X(A) \in \wp(X^2)$ as $x[\sqcup_X(A)]y$ if and only if $\forall \curvearrowright \in A, x \curvearrowright^* y$;

- $\varepsilon^{\mathscr{G}} = \emptyset \in \wp(\{1\}^2)$;

- for any set $X$, $Y$, $\forall \curvearrowright_X \in \mathscr{G}_X$, $\curvearrowright_Y \in \mathscr{G}_Y$, we define $\curvearrowright_X \otimes_{\mathscr{G}} \curvearrowright_Y \in \mathscr{G}_{X,Y}$ as $a[\curvearrowright_X \otimes_{\mathscr{G}} \curvearrowright_Y]b$ if and only if either $(a,b) \in X^2$ and $a \curvearrowright_X b$, or $(a,b) \in Y^2$ and $a \curvearrowright_Y b$.

- for any set $X$, $Y$, $\forall \curvearrowright_X \in \mathscr{G}_X$, we define $\Pi^Y_{\mathscr{G}}(\curvearrowright_X) \in \mathscr{G}_Y$ as $a[\Pi^Y_{\mathscr{G}}(\curvearrowright_X)]b$ if and only if $(a,b) \in Y^2$ and $a \curvearrowright^*_X b$.

Roughly speaking, the partial order is the opposite of the constraint set inclusion because, the more constraints, the fewer solutions. The concretization of a graph is the set of all the tuples the components of which satisfy the equality relations described by the edges of this graph. The representation of the empty word is just a graph with one vertex. Gathering some abstract elements consists in intersecting the constraint sets they are described with. Abstract join just makes the union of several graphs. Projection consists in restricting the set of the vertice, keeping equality relations on the remaining vertice. Before applying union and projection, we must close graphs, if not we may lose information.

Then we lift each domain $\mathscr{G}_X$ by adding an extra element $\bot^{\mathscr{G}}_X$. We denote $\overline{\mathscr{G}_X}$ the set $\mathscr{G}_X \cup \bot^{\mathscr{G}}_X$. The element $\bot^{\mathscr{G}}_X$ is the least element of the domain $\overline{\mathscr{G}_X}$. Moreover, the concretization of $\bot^{\mathscr{G}}_X$ is the empty set. Abstract union is lifted as follows:

$$\sqcup^{\overline{\mathscr{G}}}_X(A) = \begin{cases} \bot^{\mathscr{G}}_X & \text{if } A \subseteq \{\bot^{\mathscr{G}}_X\} \\ \sqcup^{\mathscr{G}}_X A \setminus \{\bot^{\mathscr{G}}_X\} & \text{otherwise.} \end{cases}$$

Abstract intersection is lifted as follows:

$$\sqcap^{\overline{\mathscr{G}}}_X(A) = \begin{cases} \bot^{\mathscr{G}}_n & \text{if } \bot^{\mathscr{G}}_X \in A \\ \sqcap^{\mathscr{G}}_X A & \text{otherwise.} \end{cases}$$

All other abstract primitives are lifted to be strict, which means that they return the well-typed good element of the form $\bot^{\mathscr{G}}_X$ as soon as one of their arguments is of the form $\bot^{\mathscr{G}}_X$.

We instantiate our abstract domains. We set $\mathscr{M}^\sharp_1 = \overline{\mathscr{G}}_{\{1\}}$, $\mathscr{M}^\sharp_2 = \overline{\mathscr{G}}_{\{1;2\}}$ and $\mathscr{M}^\sharp_X = \overline{\mathscr{G}}_X$ for each finite set $X$. We take $\gamma^{\mathscr{M}}_1 = \gamma^{\overline{\mathscr{G}}}_{\{1\}}$, $\gamma^{\mathscr{M}}_2 = \gamma^{\overline{\mathscr{G}}}_{\{1;2\}}$, $\gamma^{\mathscr{M}}_X = \gamma^{\overline{\mathscr{G}}}_X$.

Then the abstract primitives are defined as follows:

1. $\bot^{\mathscr{M}}_1 = \bot^{\mathscr{G}}_{\{1\}}$, $\bot^{\mathscr{M}}_2 = \bot^{\mathscr{G}}_{\{2\}}$, $\bot^{\mathscr{M}}_X = \bot^{\mathscr{G}}_X$, for any finite set $X$;

2. $\emptyset^\sharp_{\mathscr{M}} = \emptyset$;

3. $\top^{\mathscr{M}}_1 = \emptyset$;

4. $\varepsilon^{\mathscr{M}}_1 = \emptyset$;

5. $\sqcup_1^{\mathscr{M}} = \sqcup_{\{1\}}^{\overline{\mathscr{G}}}$, $\sqcup_2^{\mathscr{M}} = \sqcup_{\{1;2\}}^{\overline{\mathscr{G}}}$, $\sqcup_X^{\mathscr{M}} = \sqcup_X^{\overline{\mathscr{G}}}$, for any finite set $X$,

6. $\sqcap_1^{\mathscr{M}} = \sqcap_{\{1\}}^{\overline{\mathscr{G}}}$, and $\sqcap_2^{\mathscr{M}} = \sqcap_{\{1;2\}}^{\overline{\mathscr{G}}}$;

7. since domains are height bounded, we define $a \nabla_i^{\mathscr{M}} b = \sqcup_i^{\mathscr{M}} \{a; b\}$, for any $i \in \{1;2\}$;

8. $\text{PAIR}(a,b) = \begin{cases} \bot_{\{1;2\}}^{\mathscr{G}} & \text{if } \bot_{\{1\}}^{\mathscr{G}} \in \{a;b\}, \\ \emptyset & \text{otherwise;} \end{cases}$

9. $\text{DIAG}(a) = \begin{cases} \bot_{\{1;2\}}^{\mathscr{G}} & \text{if } \bot_{\{1;2\}}^{\mathscr{G}} \in \{a;b\}, \\ \{(1;2)\} & \text{otherwise;} \end{cases}$

10. for any set $X, Y$, any $q \in X \to Y$, and any $a \in \overline{\mathscr{G}}_X$, we define $\text{QUOTIENT}(q,a)$ as: $\begin{cases} \bot_Y^{\mathscr{G}} & \text{if } a = \bot_X^{\overline{\mathscr{G}}}, \\ \{(q(x),q(y)) \mid (x,y) \in a,\ q(x) \neq q(y)\} & \text{otherwise;} \end{cases}$

11. $\underset{(p^k)}{\text{PUSH}}(a) = a \cap (X \setminus \{(I,1)\})^2$;

12. $a \otimes b = a \otimes^{\overline{\mathscr{G}}} b$;

13. $\text{PROJ}(Y,a) = \Pi_{\overline{\mathscr{G}}}^{Y}(a)$;

14. $\text{BUILD-SG}_x(a) = \begin{cases} \bot_{\{x\}}^{\mathscr{G}} & \text{if } a = \bot_{\{1\}}^{\mathscr{G}}, \\ \emptyset & \text{otherwise;} \end{cases}$

15. $\text{EXTRACT-SG}_x(a) = \begin{cases} \bot_{\{1\}}^{\mathscr{G}} & \text{if } a = \bot_X^{\mathscr{G}}, \\ \emptyset & \text{otherwise;} \end{cases}$

16. $\text{BUILD-PR}_{x,y}(a) = \begin{cases} \bot_{\{x;y\}}^{\mathscr{G}} & \text{if } a = \bot_{\{1;2\}}^{\mathscr{G}}, \\ \{(x,y)\} & \text{if } (1,2) \in a \text{ or } (2,1) \in a, \\ \emptyset & \text{otherwise;} \end{cases}$

17. $\text{EXTRACT-PR}_{x,y}(a) = \begin{cases} \bot_{\{1;2\}}^{\mathscr{G}} & \text{if } a = \bot_{\{x;y\}}^{\mathscr{G}}, \\ \{(1,2)\} & \text{if } (x,y) \in a \text{ or } (y,x) \in a, \\ \emptyset & \text{otherwise;} \end{cases}$

The type system that is proposed in [16, 15] uses the notion of group. Groups gather some values that have been declared by the same recursive instance of a thread. Then, the type system ensures that the values of one group may not be communicated to the variables of the other groups. This ensures information confinement even inside recursive instances. However, the type system may not validate a system where a given value first exits the scope of the thread instance which has declared it and then enters again the recursive instance which has declared it. The same way, we cannot deal precisely with values that first exits the scope of the thread that has declared them, then enter again this scope with our abstract domain. The main problem is that we can only propagate equality relations. When the value is communicated to a thread with a distinct marker, we loose all information. Then, if the value is returned to the thread that has previously declared this value, we cannot infer the equality relation. To achieve this goal we need an algebraic comparison between markers. That is the purpose of Sect. 8.2.5.3.

#### 8.2.5.3 Non-uniform analysis with algebraic comparisons

We now propose an abstract domain which deals with abstract algebraic comparisons between markers. Following Prop. 4.5.18 on page 69, we only abstract each tree by the word that is obtained by following the second child of each node. We choose $m \in \{1;2\}$, according to the chosen simplification function $\phi_1$ or $\phi_2$[7]. We define the alphabet $\Sigma_m$ as $\mathscr{L}^*$ when $m = 1$ and by $\mathscr{L}$ when $m = 2$. The function $\sigma_m$ maps each tuple of at least two program points into a letter of $\sigma_m$. It is defined by $\sigma_m((a_i)) = (a_i)$ when $m = 1$ and $\sigma_m((a_i)) = a_2$ when $m = 2$. We use a reduced product between two abstractions. The reduction operators are given in Sect. 8.2.5.3.3. Our first abstraction consists in abstracting component-wise the shape of the markers associated to threads. The second one infers a comparison between the Parikh's vectors of the markers.

**8.2.5.3.1 Regular approximation.** We approximate the marker shape in regular languages. For the sake of efficiency, we only use the regular languages which may be described by a set of initial letters, a set of last letters and a succession relation between letters. The result is an efficient abstract domain of languages, the height of which is quadratic in the cardinal of the alphabet $\Sigma_m$. Moreover, abstract primitives may be computed with a $\mathscr{O}(|\Sigma_m|^2)$ worst-case time cost.

We introduce the set $\text{Reg}_{\Sigma_m}$ of tuples $(i, f, t, b)$ such that $i, f \in \wp(\Sigma_m)$, $t \in (\Sigma_m \rightarrow \wp(\Sigma_m))$ and $b \in \{0;1\}$. Each element $(i, f, t, b) \in \text{Reg}_{\Sigma_m}$ is related to a

[7]Both functions have been introduced in Prop. 4.5.18.

language on $\Sigma_m$ via a concretization function $\gamma^{\text{Reg}}_{\Sigma_m}$ defined as follows:

$$u \in \gamma^{\text{Reg}}_{\Sigma_m}(i, f, t, b) \iff \begin{cases} |u| > 0 \implies u_1 \in i \\ |u| > 0 \implies u_{|u|} \in f \\ \forall i \in [\![1; |u|[\![,\ |u|_{i+1} \in t(u_i) \\ |u| = 0 \implies b = 1. \end{cases}$$

Roughly speaking, $i$ is the set of the initial letters of the language words, $f$ is the set of the final letters. The set $t(a)$ is the set of letters which may immediately follow an occurrence of the letter $a$. The boolean $b$ is equal to 1 if the empty word belongs to the language.

It is worth noting that only letters that are both reachable from an initial letter and co-reachable from a final letter are insightful. We define a reduction operator $\rho^{\text{Reg}} \in \text{Reg}_{\Sigma_m} \to \text{Reg}_{\Sigma_m}$ that removes useless letter. Let $(i, f, t, b) \in \text{Reg}_{\Sigma_m}$ be an abstract element. Let $X \subseteq \Sigma_m$ be the set of the letters $\lambda$ that may occur in a word $u \in \gamma^{\text{Reg}}_{\Sigma_m}((i, f, t, b))$ (i.e. such that state corresponding to $\lambda$ is both reachable and co-reachable). The element $\rho^{\text{Reg}}((i, f, t, b))$ is defined as $(i \cap X, f \cap X, [\lambda \to t(\lambda) \cap X], b)$. A constructive definition of $\rho^{\text{Reg}}$ is easily deduced by using Dijkstra's shortest path closure algorithm.

**Proposition 8.2.5.** *The reduction operator satisfies the soundness property* $\gamma^{\text{Reg}}_{\Sigma_m}(a) \subseteq \gamma^{\text{Reg}}_{\Sigma_m}(\rho^{\text{Reg}}(a))$.

Then the abstract domain $\text{Reg}_{\Sigma_m}$ is fitted with a complete lattice structure $(\text{Reg}_{\Sigma_m}, \sqsubseteq^{\text{Reg}}_{\Sigma_m}, \sqcap^{\text{Reg}}_{\Sigma_m}, \sqcup^{\text{Reg}}_{\Sigma_m}, \bot^{\text{Reg}}_{\Sigma_m}, \top^{\text{Reg}}_{\Sigma_m})$ as follows:

- $\sqsubseteq^{\text{Reg}}_{\Sigma_m}$ and $\sqcup^{\text{Reg}}_{\Sigma_m}$ are defined component-wise from the usual set operations $\subseteq$ and $\cup$;
- We compute the reduction operator $\rho^{\text{Reg}}$ after each abstract intersection. Let $A = \{(i_k, f_k, t_k, b_k) \mid k \in K\}$ be set of elements in $\text{Reg}_{\Sigma_m}$, we define the element $(\sqcap^{\text{Reg}}_{\Sigma_m} A) \in \text{Reg}_{\Sigma_m}$ as $\rho^{\text{Reg}}(i', f', t', b')$, where:
  1. $i' = \cap\{i_k \mid k \in K\}$,
  2. $f' = \cap\{f_k \mid k \in K\}$,
  3. $t' = [\lambda \to \cap\{t_k(\lambda) \mid k \in K\}]$,
  4. $b' = min\{b_k \mid k \in K\}$;
- $\bot^{\text{Reg}}_{\Sigma_m} = (\emptyset, \emptyset, [\lambda \mapsto \emptyset], 0)$;
- $\top^{\text{Reg}}_{\Sigma_m} = (\Sigma_m, \Sigma_m, [\lambda \mapsto \Sigma_m], 1)$.

Furthermore, the language that contains only the empty word is described by the tuple $(\emptyset,\emptyset,[\lambda\mapsto\emptyset],1)$ and is denoted by $\varepsilon^{\mathrm{Reg}}_{\Sigma_m}$. At last, we can define the primitive $\textsc{push}^{\mathrm{Reg}}_{\Sigma_m} : \mathrm{Reg}_{\Sigma_m}\times\Sigma_m\to\mathrm{Reg}_{\Sigma_m}$ which adds a letter at the end of all the words of a language by $\textsc{push}^{\mathrm{Reg}}_{\Sigma_m}((i,f,t,b),\lambda)=(i',f',t',b')$ where:

$$\begin{cases} i'=\begin{cases} i\cup\{\lambda\} & \text{if } b=1\\ i & \text{otherwise;}\end{cases}\\ f'=\begin{cases}\{\lambda\} & \text{if } b=1 \text{ or } i\neq\emptyset\\ \emptyset & \text{otherwise;}\end{cases}\\ t'=\left[a\mapsto\begin{cases} t(a)\cup\{\lambda\} & \text{if } a\in f\\ t(a) & \text{otherwise}\end{cases}\right];\\ b'=0.\end{cases}$$

We instantiate our abstract domains. We set $\mathscr{M}^\sharp_1=\mathrm{Reg}_{\Sigma_m}$, $\mathscr{M}^\sharp_2=(\mathrm{Reg}_{\Sigma_m})^2$ and $\mathscr{M}^\sharp_X=X\to\mathrm{Reg}_{\Sigma_m}$, for any finite set $X$. We take $\gamma^{\mathscr{M}}_1=\gamma^{\mathrm{Reg}}_{\Sigma_m}$. The concretization function $\gamma^{\mathscr{M}}_2$ applies the function $\gamma^{\mathrm{Reg}}_{\Sigma_m}$ pair-wise. For any finite set $X$ the concretization function $\gamma^{\mathscr{M}}_X$ applies the function $\gamma^{\mathrm{Reg}}_{\Sigma_m}$ component-wise.

Then the abstract primitives are defined as follows:

1. $\bot^{\mathscr{M}}_1=\bot^{\mathrm{Reg}}_{\Sigma_m}$; $\bot^{\mathscr{M}}_2=(\bot^{\mathrm{Reg}}_{\Sigma_m},\bot^{\mathrm{Reg}}_{\Sigma_m})$; $\bot^{\mathscr{M}}_X=[x\mapsto\bot^{\mathrm{Reg}}_{\Sigma_m}]$;

2. $\emptyset^\sharp_{\mathscr{M}}$ is the function $\emptyset_{\mathrm{Reg}_{\Sigma_m}}$;

3. $\top^{\mathscr{M}}_1=\top^{\mathrm{Reg}}_{\Sigma_m}$;

4. $\varepsilon^{\mathscr{M}}_1=\varepsilon^{\mathrm{Reg}}_{\Sigma_m}$;

5. $\sqcup^{\mathscr{M}}_1=\sqcup^{\mathrm{Reg}}_{\Sigma_m}$, $\sqcup^{\mathscr{M}}_2$ applies $\sqcup^{\mathrm{Reg}}_{\Sigma_m}$ pair-wise, and for any finite set $X$, $\sqcup^{\mathscr{M}}_X$ applies $\sqcup^{\mathrm{Reg}}_{\Sigma_m}$ component-wise;

6. $\sqcap^{\mathscr{M}}_1=\sqcap^{\mathrm{Reg}}_{\Sigma_m}$, and $\sqcap^{\mathscr{M}}_2$ applies $\sqcap^{\mathrm{Reg}}_{\Sigma_m}$ pair-wise;

7. since domains are height bounded, we set $a\nabla^{\mathscr{M}}_i b=\sqcup^{\mathscr{M}}_i\{a;b\}$, for any $i\in\{1;2\}$;

8. $\textsc{pair}(a,b)=(a,b)$;

9. $\textsc{diag}((a,b))=(\sqcap^{\mathscr{M}}_1\{a;b\},\sqcap^{\mathscr{M}}_1\{a;b\})$;

10. for any set $X$, $Y$, for any $q \in X \to Y$, and for any $a \in \mathrm{Reg}_{\Sigma_m}$, we define $\textsc{quotient}(q,a)$ as:

$$\begin{cases} \bot^{\mathscr{M}}_{Y} & \text{if } a = \bot^{\mathscr{M}}_{X} \\ \bot^{\mathscr{M}}_{Y} & \text{if } \exists y \in Y,\ \sqcap^{\mathrm{Reg}}_{\Sigma_m}\{a(x) \mid q(x) = y\} = \bot^{\mathrm{Reg}}_{\Sigma_m} \\ [y \mapsto \sqcap^{\mathrm{Reg}}_{\Sigma_m}\{a(x) \mid q(x) = y\}] & \text{otherwise;} \end{cases}$$

11. $\textsc{push}_{(p^k)}(a)$ maps the variable $(I,1)$ to the abstract element $push^{\mathrm{Reg}}(a((I,2)),\sigma_m((p^k)))$ and any other variables $X$ to the element $a(X)$;

12. 
    - in the case when there exists $x \in Dom(a) \cap Dom(b)$ such that $\bigcap^{\mathrm{Reg}}\{a(x),b(x)\} = \bot^{\mathrm{Reg}}$, we set:

$$a \otimes b = \bot^{\mathscr{M}}_{Dom(a)\cup Dom(b)},$$

    - otherwise, we set:

$$a \otimes b = \begin{bmatrix} x \mapsto a(x) & \text{if } x \in Dom(a) \setminus Dom(b) \\ x \mapsto b(x) & \text{if } x \in Dom(b) \setminus Dom(a) \\ x \mapsto \sqcap^{\mathrm{Reg}}_{\Sigma_m}\{a(x),b(x)\} & \text{if } x \in Dom(a) \cap Dom(b) \end{bmatrix};$$

13. $\textsc{proj}(Y,a) = a_{|Y}$;

14. $\textsc{build-sg}_x(a) = [x \mapsto a]$, $\textsc{extract-sg}_x(a) = a(x)$,
$\textsc{build-pr}_{x,y}((a,b)) = [x \mapsto a, y \mapsto b]$, and
$\textsc{extract-pr}_{x,y}(a) = (a(x),a(y))$.

**8.2.5.3.2 Numerical approximation.** Our second abstraction captures relational comparisons between the occurrence number of each pattern inside sets of marker maps. For each finite set $X$, we introduce a set $\mathscr{V}_X$ of distinct variables $\{v^{\lambda}_{x} \mid x \in X,\ \lambda \in \Sigma_m\}$. The abstract domain $\wp(\mathscr{V}_X \to \mathbb{N})$ is related to $\wp(X \to \mathscr{M})$ by the monotonic map $\gamma^{\mathrm{Rel}}_{X}$:

$$\gamma^{\mathrm{Rel}}_{X}(A) = \left\{ f \in X \to \mathscr{M} \,\middle|\, \exists w \in A,\ \forall v^{\lambda}_{x} \in \mathscr{V}_X,\ w(v^{\lambda}_{x}) = |\phi_m(f(x))|_{\lambda} \right\}.$$

Then the power set $\wp(\mathscr{V}_X \to \mathbb{N})$ is related to a numerical domain. Many relational numerical domains have been introduced in the literature [47, 26, 42, 55]. We propose two choices according to the expected trade-off between complexity and accuracy. They both use the complete lattice of affine equality systems among a set of variables. This domain is described with its lattice operations $(\sqsubseteq^{\mathscr{K}}_{\mathscr{V}}, \sqcup^{\mathscr{K}}_{\mathscr{V}}, \perp^{\mathscr{K}}_{\mathscr{V}}, \sqcap^{\mathscr{K}}_{\mathscr{V}}, \top^{\mathscr{K}}_{\mathscr{V}})$ in [47]. Given a set of variables $\mathscr{V}$, we denote by $\mathscr{K}_{\mathscr{V}}$ the domain of the affine equality systems among the elements of $\mathscr{V}$. This domain is related to the power set $\wp(\mathscr{V} \to \mathbb{N})$ by a concretization function that we denote $\gamma^{\mathscr{K}}_{\mathscr{V}}$.

**8.2.5.3.2.1 Component-wise affine comparison** The first abstraction choice consists in relating each letter $\lambda$ in the alphabet $\Sigma_m$ with affine relationships between the occurrence number of $\lambda$ in each marker. For any set $X$ of variables (each variable is associated with a marker), we introduce $Id^{\mathrm{Rel}}_X$ as the complete lattice $(\Sigma_m \to \mathscr{K}_{\{v_x \mid x \in X\}})$ defined point-wise.

We instantiate our abstract domains. We set $\mathscr{M}^\sharp_1 = Id^{\mathrm{Rel}}_{\{1\}}$, $\mathscr{M}^\sharp_2 = Id^{\mathrm{Rel}}_{\{1,2\}}$ and $\mathscr{M}^\sharp_X = Id^{\mathrm{Rel}}_X$, for any finite set $X$. Concretizations are defined as follows:

- $\gamma^{\mathscr{M}}_1(f) = \gamma^{\mathrm{Rel}}_{\{1\}}\left(\left\{g \in \mathscr{V}_{\{1\}} \to \mathbb{N} \;\middle|\; \forall \lambda \in \Sigma_m,\ [v_1 \to g(v^\lambda_1)] \in \gamma^{\mathscr{K}}_{\{v_1\}}(f(\lambda))\right\}\right)$,
- $\gamma^{\mathscr{M}}_2(f) = \gamma^{\mathrm{Rel}}_{\{1;2\}}\left(\left\{g \middle| \forall \lambda \in \Sigma_m,\ [v_i \to g(v^\lambda_i), \forall i \in \{1;2\}] \in \gamma^{\mathscr{K}}_{\{v_1,v_2\}}(f(\lambda))\right\}\right)$,
- and $\gamma^{\mathscr{M}}_X(f) = \gamma^{\mathrm{Rel}}_X\left(\left\{g \;\middle|\; \forall \lambda \in \Sigma_m,\ [v_x \to g(v^\lambda_x), \forall x \in X] \in \gamma^{\mathscr{K}}_X(f(\lambda))\right\}\right)$, for any finite set $X$.

Then the abstract primitives are defined as follows:

1. the function $\perp^{\mathscr{M}}_1$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\perp^{\mathscr{K}}_{\{v_1\}}$; the function $\perp^{\mathscr{M}}_2$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\perp^{\mathscr{K}}_{\{v_1;v_2\}}$; the function $\perp^{\mathscr{M}}_X$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\perp^{\mathscr{K}}_{\{v_x \mid x \in X\}}$;
2. the function $\emptyset^\sharp_{\mathscr{M}}$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\top^{\mathscr{K}}_{\emptyset}$;
3. the function $\top^{\mathscr{M}}_1$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\top^{\mathscr{K}}_{\{v_1\}}$;
4. $\varepsilon^{\mathscr{M}}_1$ maps each letter $\lambda \in \Sigma_m$ into the affine system $\{v_1 = 0$ in $\mathscr{K}_{\{v_1\}}$;
5. $\sqcup^{\mathscr{M}}_1$, $\sqcup^{\mathscr{M}}_2$, and $\sqcup^{\mathscr{M}}_X$ applies $\sqcup^{\mathscr{K}}_{\{v_1\}}$, $\sqcup^{\mathscr{K}}_{\{v_1;v_2\}}$, and $\sqcup^{\mathscr{K}}_{\{v_x \mid x \in X\}}$ component-wise.
6. $\sqcap^{\mathscr{M}}_1$ and $\sqcap^{\mathscr{M}}_2$ applies $\sqcap^{\mathscr{K}}_{\{v_1\}}$ and $\sqcap^{\mathscr{K}}_{\{v_1;v_2\}}$ component-wise. Then, if all components are distinct from $\perp^{\mathscr{K}}_{\{v_1\}}$ (resp. $\perp^{\mathscr{K}}_{\{v_1;v_2\}}$), the result is kept like this, otherwise it is replaced with $\perp^{\mathscr{M}}_1$ (resp. $\perp^{\mathscr{M}}_2$). It is the usual so called *coalescent product*.

7. since domains are height bounded, we set $a\nabla_i^{\mathscr{M}} b = \sqcup_i^{\mathscr{M}}\{a;b\}$, for any $i \in \{1;2\}$;

8. $\textsc{pair}(a,b)$ maps each letter $\lambda \in \Sigma_m$ to the system composed of the constraints in $a(\lambda)$ and the constraints in $b(\lambda)$, where each occurrence of the variable $v_1$ in a constraint of $b(\lambda)$ is replaced with the variable $v_2$;

9. $\textsc{diag}(a)$ maps each letter $\lambda \in \Sigma_m$ to the system that contains both the constraints in the system $a(\lambda)$ and the constraint $v_1 = v_2$;

10. for any set $X, Y$, for any $q \in X \to Y$, the function $\textsc{quotient}(q,a)$ associates each letter $\lambda \in \Sigma_m$ to the system that is obtained by replacing each occurrence of the variable $v_x$ (where $x \in X$) with the variable $v_{q(x)}$ in the system $a(\lambda)$.

11. the function $\underset{(p^k)}{\textsc{push}}(a)$ maps each letter $\lambda \in \Sigma_m$ to the system that is obtained by first removing any constraint about the variable $v_{(I,1)}$ in the system $a(\lambda)$ (by using Gaussian elimination), and by then inserting the constraint $v_{(I,1)} = v_{(I,2)} + \delta^{\lambda}_{\sigma_m((p^k))}$ (where $\delta^x_y$ is either equal to 1 if x=y, or equal to 0 otherwise);

12. for any set $X$ and $Y$, for any $a \in Id_X^{\mathrm{Rel}}$ and any $b \in Id_Y^{\mathrm{Rel}}$, the function $a \otimes b$ maps each letter $\lambda \in \Sigma_m$ to the system that is obtained by first collecting all the constraints in $a(\lambda)$ and all the constraints in $b(\lambda)$, and then performing Gaussian elimination[8]. Moreover, if there exists one letter such that the obtained system is $\bot^{\mathscr{K}}_{\{v_x \mid x \in X \cup Y\}}$not satisfiable, we associate each letter $\lambda \in \Sigma_m$ with the system $\bot^{\mathscr{K}}_{\{v_x \mid x \in X \cup Y\}}$.

13. $\textsc{proj}(Y,a)$ maps each letter $\lambda \in \Sigma_m$ to the system $a(\lambda)$ in where we have used Gaussian elimination to collect all the constraints involving only variables in $\{v_y \mid y \in Y\}$;

14. For any letter $\lambda \in \Sigma_m$,

    (a) the affine system $\textsc{build-sg}_x(a)(\lambda)$ is obtained by replacing any occurrence of the variable $v_1$ with the variable $v_x$ in the affine system $a(\lambda)$;

    (b) the affine system $\textsc{extract-sg}_x(a)(\lambda)$ is obtained by first collecting (*via* Gaussian elimination) all the constraints in $a(\lambda)$ that only constrain the variable $v_x$, and then by replacing $v_x$ with $v_1$ in those constraints;

[8]It is worth noting that this is not an affi ne intersection since the system $a(\lambda)$ does not tie the variables $v_x$ with $x \in Y \setminus X$ and since the system $b(\lambda)$ does not tie the variables $v_x$ with $x \in X \setminus Y$.

(c) the affine system $\text{BUILD-PR}_{x,y}(a)(\lambda)$ is obtained by replacing any occurrence of the variable $v_1$ with the variable $v_x$ and any occurrence of the variable $v_2$ with the variable $v_y$ in the affine system $a(\lambda)$;

(d) the affine system $\text{EXTRACT-PR}_{x,y}(a)(\lambda)$ is obtained by first collecting (*via* Gaussian elimination) all the constraint in $a(\lambda)$ that only constrains the variables $v_x$ and $v_y$, and then by replacing the variable $v_x$ with the variable $v_1$ and the variable $v_y$ with the variable $v_2$ in those constraints;

**8.2.5.3.2.2 Global affine comparison** The second choice consists in abstracting globally all constraints. Roughly speaking, the variable $v_x^\lambda$ denotes the occurrence number of $\lambda$ in the image of $x$. For any set $X$ of variables (each variable is associated with a marker), We introduce $Id_X^{\text{Rel}}$ as the complete lattice $\mathscr{K}_{\{v_x^\lambda \mid x\in X,\, \lambda\in\Sigma_m\}}$.

We instantiate our abstract domains. We set $\mathscr{M}_1^\sharp = Id_{\{1\}}^{\text{Rel}}$, $\mathscr{M}_2^\sharp = Id_{\{1,2\}}^{\text{Rel}}$ and $\mathscr{M}_X^\sharp = Id_X^{\text{Rel}}$, for any finite set $X$. Concretizations are defined as follows:

- $\gamma_1^{\mathscr{M}} = \gamma_{\{1\}}^{\text{Rel}} \circ \gamma_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}^{\mathscr{K}}$,
- $\gamma_2^{\mathscr{M}} = \gamma_{\{1;2\}}^{\text{Rel}} \circ \gamma_{\{v_i^\lambda \mid \lambda\in\Sigma_m,\, i\in\{1;2\}\}}^{\mathscr{K}}$,
- and $\gamma_X^{\mathscr{M}} = \gamma_X^{\text{Rel}} \circ \gamma_{\{v_x^\lambda \mid \lambda\in\Sigma_m,\, x\in X\}}^{\mathscr{K}}$,

Then the abstract primitives are defined as follows:

1. we define the bottom element $\perp_1^{\mathscr{M}}$ as the affine system $\perp_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}^{\mathscr{K}}$, the bottom element $\perp_2^{\mathscr{M}}$ by the affine system $\perp_{\{v_i^\lambda \mid \lambda\in\Sigma_m,\, i\in\{1;2\}\}}^{\mathscr{K}}$, and the bottom element $\perp_X^{\mathscr{M}}$ by the affine system $\perp_{\{v_x^\lambda \mid \lambda\in\Sigma_m,\, x\in X\}}^{\mathscr{K}}$;

2. we define the empty function abstraction $\emptyset_{\mathscr{M}}^\sharp$ as the affine system $\top_\emptyset^{\mathscr{K}}$;

3. we define the top element $\top_1^{\mathscr{M}}$ as the affine system $\top_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}^{\mathscr{K}}$;

4. the abstraction of the empty marker $\varepsilon_1^{\mathscr{M}}$ is defined as the affine system $\left(\{v_1^\lambda = 0,\, \forall\lambda\in\Sigma_m\right)$ in $\mathscr{K}_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}$;

5. the operators $\sqcup_1^{\mathscr{M}}$, $\sqcup_2^{\mathscr{M}}$, and $\sqcup_X^{\mathscr{M}}$ are respectively defined as $\sqcup_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}^{\mathscr{K}}$, $\sqcup_{\{v_i^\lambda \mid \lambda\in\Sigma_m,\, i\in\{1;2\}\}}^{\mathscr{K}}$, and $\sqcup_{\{v_x^\lambda \mid \lambda\in\Sigma_m,\, x\in X\}}^{\mathscr{K}}$.

6. $\sqcap_1^{\mathscr{M}}$ and $\sqcap_2^{\mathscr{M}}$ applies $\sqcap^{\mathscr{K}}_{\{v_1^\lambda \mid \lambda\in\Sigma_m\}}$ and $\sqcap^{\mathscr{K}}_{\{v_i^\lambda \mid \lambda\in\Sigma_m,\, i\in\{1;2\}\}}$.

7. since domains are height bounded, we set $a\nabla_i^{\mathscr{M}}b$ as $\sqcup_i^{\mathscr{M}}\{a;b\}$, for any $i \in \{1;2\}$;

8. the affine system $\textsc{pair}(a,b)$ is computed by collecting all the constraints that are either in the system $a$, or in the system $b$, after having replaced each occurrence of a variable that matches $v_1^\lambda$ in the system $b$ with the variable $v_2^\lambda$;

9. the affine system $\textsc{diag}(a)$ contains both the constraints in the affine system $a$ and each constraint $v_1^\lambda = v_2^\lambda$, for any $\lambda \in \Sigma_m$;

10. for any set $X$, $Y$, for any $q \in X \to Y$, the affine system $\textsc{quotient}(q,a)$ is obtained by replacing each occurrence of a variable $v_x^\lambda$ (where $x \in X$) with the variable $v_{q(x)}^\lambda$ in the system $a$.

11. the affine system $\underset{(p^k)}{\textsc{push}}(a)$ is obtained by first removing any constraint about the variables $v_{(I,1)}^\lambda$ in the system $a$ by using Gaussian elimination, and then by inserting the constraint $v_{(I,1)}^\lambda = v_{(I,2)}^\lambda + \delta^\lambda_{\sigma_m((p^k))}$ for any $\lambda$ in $\Sigma_m$ (where $\delta_y^x$ is either equal to 1 if x=y, or equal to 0 otherwise);

12. for any set $X$ and $Y$, for any $a \in Id_X^{\mathrm{Rel}}$ and any $b \in Id_Y^{\mathrm{Rel}}$, the system $a \otimes b$ is obtained by first collecting all the constraints in $a$ and all the constraints in $b$, and by using Gaussian elimination. It is worth noting that this is not an affine intersection since the system $a$ does not tie the variables of the form $v_x^\lambda$ with $x \in Y \setminus X$ and since the system $b$ does not tie the variables $v_x^\lambda$ with $x \in X \setminus Y$.

13. the affine system $\textsc{proj}(Y,a)$ is given by the system $a$ in where we have used Gaussian elimination to collect all the constraints that only involve variables in $\{v_y^\lambda \mid \lambda \in \Sigma_m,\, y \in Y\}$;

14. (a) the affine system $\textsc{build-sg}_x(a)$ is obtained by replacing, for each $\lambda \in \Sigma_m$, any occurrence of the variable $v_1^\lambda$ with the variable $v_x^\lambda$ in the affine system $a$;

    (b) the affine system $\textsc{extract-sg}_x(a)$ is obtained by first collecting (*via* Gaussian elimination) all the constraints in $a$ that only constrain the variables of the form $v_x^\lambda$, and then by replacing, for each $\lambda \in \Sigma_m$, the variable $v_x^\lambda$ with the variable $v_1^\lambda$ in those constraints;

(c) the affine system $\text{BUILD-PR}_{x,y}(a)$ is obtained by replacing, for each $\lambda \in \Sigma_m$, any occurrence of the variable $v_1^\lambda$ with the variable $v_x^\lambda$ and any occurrence of the variable $v_2^\lambda$ with the variable $v_y^\lambda$ in the affine system $a$;

(d) the affine system $\text{EXTRACT-PR}_{x,y}(a)$ is obtained first by collecting (*via* Gaussian elimination) all the constraints in $a$ that only constrain the variables of the form $v_x^\lambda$ and $v_y^\lambda$, and then by replacing, for each $\lambda \in \Sigma_m$, each occurrence of the variable $v_x^\lambda$ with the variable $v_1^\lambda$ and each occurrence of the variable $v_y^\lambda$ with the variable $v_2^\lambda$ in those constraints.

**8.2.5.3.3 Reduced product** We use a reduced product between the regular abstraction and one of the two relational abstractions to propagate the fact that a given letter does not occur inside a given marker:

1. We use the regular approximation to detect an upper-set of the letters that occur in each marker (we know that occurring letters are both reachable from an initial letter and co-reachable from a final letter). Then this upper-set is used to insert new affine constraints which encodes that these letters do not occur.

2. We use affine constraints to detect which variables are equal to 0. We remove the corresponding vertice in graphs.

Iterating these reduction steps define a lower closure operator $\rho$ (i.e. $\rho$ is monotonic, anti-extensive, and idempotent) over the product of the two marker abstractions. Moreover, the operator $\rho$ is sound (i.e. $\gamma^{\text{SHAPE}}(a) \cap \gamma^{\text{Rel}}(b) \subseteq \gamma^{\text{SHAPE}}(\mathit{fst}(\rho(a,b)) \cap \gamma^{\text{Rel}}(\mathit{snd}(\rho(a,b))))$. So it can be used as a reduction operator.

## 8.2.6 Prototypes and analysis examples

### 8.2.6.1 Two prototypes

We now present some examples of analysis result. These results have been automatically computed by using the $\pi$-s.a prototype [34] for the systems that are written in the $\pi$-calculus, and the *amb*-s.a. prototype [33] for the systems that are written in the *ambient* calculus. These two prototypes belong to preliminary works, since they do not work at the meta language level.

They propose the choice between three analyses: the uniform one, the non-uniform one with the marker abstraction $\phi_2$, and the non-uniform one with the marker abstraction $\phi_1$. Non-uniform analysis consists in the product between the regular and the relational analyses. Confinement analysis has not been implemented yet. Relational analysis used the component-wise abstraction. Global comparison that was first used in previous version turns out to be too costly to scale up. Complexity results are provided in [32]. These two prototypes can be used on line.

#### 8.2.6.2 Examples

We now describe results obtained on our examples. All these results are obtained by using the marker abstraction $\phi_2$. In the description of these results, we make no distinction between a marker and its abstraction by $\phi_2$.

**Example 8.2.6.** *In the* ftp-*server (Cf. Ex. 2.1.1 on page 18), the analyzer proves that the name of a channel opened by the restriction* $(\nu\ \boldsymbol{address})$ *may only be communicated to the variable **email** or to the variable **address**, and that the name of a channel opened by the restriction* $(\nu\ \boldsymbol{request})$ *may only be communicated to the variable **request**, to the variable **data** or to the variable **rep**. More specifically, it discovers that each time a thread* $\boldsymbol{email}![\boldsymbol{rep}]$ *is spawned, there exist p,q in* $\mathbb{N}$ *such that the thread marker is* $(1,16).(1,5)^p.((2,4).(6,3).(2,12))^q.(2,4).(6,3)$*; the variable **email** is linked to the name of a channel opened by the restriction* $(\nu\ \boldsymbol{address})$ *of the instance the marker of which was* $(1,16).(1,5)^p$*; and the variable **rep** is linked to the name of a channel opened by the restriction* $(\nu\ \boldsymbol{request})$ *of the instance the marker of which was* $(1,16).(1,5)^p$*. This is enough to prove that both variables **email** and **data** are linked to names of channels opened by the same instance of the client resource and so the answer to a query may only be sent back to the correct client.*

**Example 8.2.7.** *In the token-ring (Cf. Ex. 2.1.2 in page 18), the analyzer discovers that in each instance of a thread* $\mathbf{mon}![\boldsymbol{left},\boldsymbol{right}]$*, the variable **left** is either bound to the name of a channel opened by an instance of the* $(\nu\ \boldsymbol{right})$ *restriction or to the name of a channel opened by an instance of the* $(\nu\ \mathbf{left0})$ *restriction, and the variable **right** is always bound to the name of a channel opened by an instance of the* $(\nu\ \boldsymbol{right})$ *restriction. More specifically, in the case where the variable **left** is bound to the name of a channel opened by an instance of the* $(\nu\ \boldsymbol{right})$ *restriction, it discovers that there exists* $n \in \mathbb{N}$ *such that the instance marker of* $\mathbf{mon}![\boldsymbol{left},\boldsymbol{right}]$ *is* $(\mathbf{1},\mathbf{6}).(\mathbf{1},\mathbf{3})^{\mathbf{n+1}}$*; the variable **left** is linked to the name of a channel opened by the restriction* $(\nu\ \boldsymbol{right})$ *of the instance the marker of which was* $(\mathbf{1},\mathbf{6}).(\mathbf{1},\mathbf{3})^{\mathbf{n}}$*; and the variable **right** is linked to the name of a channel opened by the restriction* $(\nu\ \boldsymbol{right})$ *of the instance the marker of which was* $(\mathbf{1},\mathbf{6}).(\mathbf{1},\mathbf{3})^{\mathbf{n+1}}$*. This is enough to prove that each process may only be linked to either the next one or to the first one.*

**Example 8.2.8.** *We also run our analysis on the* ftp-*server written in the mobile ambients (Cf. Ex. 3.1.1 on page 38). The analyzer proves that an ambient name created by the binder* $(\nu\ \boldsymbol{q})$ *may only be communicated either to a thread at program point* **16** *in an ambient named* **request** *surrounded by an ambient named* ***p****, or to a thread at program point* **9** *in an ambient named* **answer** *surrounded by an ambient named* ***p****. More specifically, in the second case, we also capture that there exists an integer integer* $n \in \mathbb{N}$ *such that the marker of the thread at program point* **9** *is* $(13,24).(13,23)^n.(3,22)$*; the marker of the value that is associated to the variable rep is* $(13,24).(13,23)^n$ *in the thread at program point* 9*; the thread at program point* **9** *is enclosed in an ambient the marker of which is* $(13,24).(13,23)^n.(3,22)$*; the enclosing ambient is itself enclosed in an ambient the thread marker of which is* $(13,24).(13,23)^n$ *and the name of which is tagged with the marker* $(13,24).(13,23)^n$*. This is enough to prove that the name communicated inside the* **answer** *ambient and the name of the packet which surrounds this* **answer** *ambient have been both declared by the same recursive instance of the client resource.*

**Remark 8.2.9.** *Our confinement analysis is not simply an abstraction of our non-uniform analysis since two distinct markers may be recognized by the same automaton while containing the same occurrence number of each pattern (i.e having the same Parikh's vector [64]). The equality of the Parikh's vectors implies the equality of the markers if they are recognized by an automaton that contains only one acyclic path between an initial and a final state, without embedded cycle, and such that the set of the Parikh's vectors of the cycles of this automaton are linearly independent. Nevertheless, we may use a reduced product of both our confinement analysis and our non-uniform control flow analysis to solve this problem.*

**Remark 8.2.10.** *The uniform analysis is not complete with respect to the non-uniform one, this means, that computing the non-uniform analysis and then abstracting the result in order to ignore marker information may give more accurate results than directly computing the result of the uniform analysis. This is illustrated in example 8.2.11.*

**Example 8.2.11.** *We consider the following mobile system:*

$(\nu\ \mathrm{a})(\nu\ \mathrm{b})(\nu\ \mathrm{x})$
$(\ *x?^1[z]((\nu\ t)z!^2[t]t!^3[z])$
$|\ *\mathrm{repli}?^4[]\mathrm{x}!^5[\mathrm{a}]$
$|\ *\mathrm{repli}?^6[]\mathrm{x}!^7[\mathrm{b}]$
$|\ *\mathrm{a}?^8[i]i?^9[j]\mathrm{trace}!^{10}[j]$
$)$

*This system is composed of four resources. The second and the third ones allow the spawning of an unbounded number of threads either of the form* $\mathbf{x}!^5[\mathbf{a}]$ *or of the form* $\mathbf{x}!^7[\mathbf{b}]$*. The first resource may be replicated by a thread either of the form* $\mathbf{x}!^5[\mathbf{a}]$ *or of the form* $\mathbf{x}!^7[\mathbf{b}]$*, nevertheless the behavior of the spawned instance is deeply bound to which kind of thread has replicated the resource:*

- *when the resource is replicated with a thread of the form* $\mathbf{x}!^5[\mathbf{a}]$*: a channel is opened; its name **t** is sent via the channel named* **a***, so that **t** may be send to an instance of the fourth resource; this instance may then receive the name* **a** *via the channel denoted by **t**; then the instance of the first resource may send the name* **a** *via the channel named **t**, so that the instance of the fourth resource may receive the name* **a** *via the channel named **t** and send it through the channel named* **trace***; then an intruder may get the name* **a***, by spying the channel named* **trace***;*

- *when the resource is replicated with a thread of the form* $\mathbf{x}!^7[\mathbf{b}]$*: a channel is opened, its name **t** is sent via the channel named* **b***, but may not be received, so the instance is stuck, and no intruder may get the name* **b***.*

*The non-uniform analysis captures the fact that the name* **b** *may not be spied by an intruder, while the uniform does not. Roughly speaking, the main reason is that the non-uniform analysis relates the names communicated to a thread with the history of the replications which have led to the creation of this thread, while the uniform analysis abstracts this information away.*

### 8.2.7 Comparing these analyses

In this section, we propose sufficient conditions over a marker abstraction that ensure that the obtained abstraction is monotonic and sufficient conditions over two marker abstractions that ensure that the first corresponding abstraction locally approximates the second corresponding abstraction. This allows for the comparison of the accuracy of our analyses. It is worth noting that we compare not only the local accuracy of the transfer functions, but also the accuracy of the whole analyses.

#### 8.2.7.1 Monotonicity and approximation of marker abstraction

We consider two abstract domains $\mathscr{M}_1^\sharp$ and $\overline{\mathscr{M}}_1^\sharp$ that describe markers, two abstract domains $\mathscr{M}_2^\sharp$ and $\overline{\mathscr{M}}_2^\sharp$ that describe marker pairs, and two abstract domains $\mathscr{M}_X^\sharp$ and $\overline{\mathscr{M}}_X^\sharp$ for each finite set $X$ to describe marker functions over the domain $X$. We suppose that we are given all the primitives that are required in

Sect. 8.2.1. The abstraction that we obtain when we instantiate the generic framework in Sect. 8.2.1 with the marker domains $\mathscr{M}_1^\sharp$, $\mathscr{M}_2^\sharp$, and $(\mathscr{M}_X^\sharp))$ is denoted by $\mathscr{A}$. The same way, the abstraction that we obtain when we instantiate the generic framework in Sect. 8.2.1 with the marker domains $\overline{\mathscr{M}}_1^\sharp$, $\overline{\mathscr{M}}_2^\sharp$, and $(\overline{\mathscr{M}}_X^\sharp))$ is denoted by $\overline{\mathscr{A}}$.

**8.2.7.1.1 Monotonicity** In this section, we introduce the notion of monotonic marker abstraction. Then we relate this notion to the notion of monotonic abstraction (Cf. 7.4.1 on page 163).

**Definition 8.2.12 (monotonic marker abstraction).** We say that the marker abstraction $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ is monotonic if the following properties are satisfied:

1. $\forall i \in \{1;2\}$,
   - $\forall A, B \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp),\ A \subseteq B \implies \sqcup_i^{\mathscr{M}}(A) \sqsubseteq_i^{\mathscr{M}} \sqcup_i^{\mathscr{M}}(B)$;
   - $\forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp),\ \forall X, Y \in \mathscr{M}_i^\sharp$,
     $X \sqsubseteq_i^{\mathscr{M}} Y \implies \sqcup_i^{\mathscr{M}}(A \cup \{X\}) \sqsubseteq_i^{\mathscr{M}} \sqcup_i^{\mathscr{M}}(A \cup \{Y\})$;
2. $\forall i \in \{1;2\}$,
   - $\forall A, B \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp),\ A \subseteq B \implies \sqcap_i^{\mathscr{M}}(B) \sqsubseteq_i^{\mathscr{M}} \sqcap_i^{\mathscr{M}}(A)$;
   - $\forall A \in \wp_{\text{finite}}(\mathscr{M}_i^\sharp),\ \forall X, Y \in \mathscr{M}_i^\sharp$,
     $X \sqsubseteq_i^{\mathscr{M}} Y \implies \sqcap_i^{\mathscr{M}}(A \cup \{X\}) \sqsubseteq_i^{\mathscr{M}} \sqcap_i^{\mathscr{M}}(A \cup \{Y\})$;
3. $\forall i \in \{1;2\}$, the operator $\nabla_i^{\mathscr{M}}$ is monotonic with respect to each of its arguments;
4. the operator PAIR is monotonic with respect to each of its arguments;
5. the operator DIAG is monotonic;
6. for any finite set $X$,
   - $\forall A, B \in \wp_{\text{finite}}(\mathscr{M}_X^\sharp),\ A \subseteq B \implies \sqcup_X^{\mathscr{M}}(A) \sqsubseteq_X^{\mathscr{M}} \sqcup_X^{\mathscr{M}}(B)$;
   - $\forall A \in \wp_{\text{finite}}(\mathscr{M}_X^\sharp),\ \forall x, y \in \mathscr{M}_X^\sharp$,
     $x \sqsubseteq_i^{\mathscr{M}} y \implies \sqcup_i^{\mathscr{M}}(A \cup \{x\}) \sqsubseteq_i^{\mathscr{M}} \sqcup_i^{\mathscr{M}}(A \cup \{y\})$;
7. $\forall q \in X \to Y$, the operator $[a \to \text{QUOTIENT}(q, a)]$ is monotonic;

8. for any integer $n \geq 2$, for any variable set $X$ such that $\{(I,i) \mid 1 \leq k \leq n\} \subseteq X$, for any tuple $(p^k)$ of program point labels, the operator $[a \in \mathscr{M}_X^\sharp \to \underset{(p^k)}{\text{PUSH}}(a)]$ is monotonic;

9. the join operator is monotonic with respect to each of its arguments;

10. for any finite sets $X$ and $Y$, the operator $[a \in \mathscr{M}_X^\sharp \to \text{PROJ}(Y,a)]$ is monotonic;

11. let $X$ be a set of elements such that $X \subseteq Y$, let $x$ and $y$ be two distinct elements in $X$, the operators $[a \to \text{BUILD-SG}_x(a)]$, $[a \to \text{EXTRACT-SG}_x(a)]$, $[a \to \text{BUILD-PR}_{x,y}(a)]$, and $[a \to \text{EXTRACT-PR}_{x,y}(a)]$ are monotonic.

**Proposition 8.2.13.** *If the marker abstraction $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ is monotonic, then the analysis $\mathscr{A}$ is monotonic.*

The following properties help in compositionally proving that a marker abstraction is monotonic:

**Proposition 8.2.14 (monotonicity stability).** *The following properties are satisfied:*

1. *The Cartesian product (see Sect. 8.2.3 on page 196) of two monotonic marker abstractions is also monotonic.*

2. *The reduction (see Def. 8.2.4 on page 197) of a monotonic marker abstraction by some monotonic reduction operator is also monotonic.*

**8.2.7.1.2 Local comparison** In this section, we introduce the notion of local comparison among marker abstractions. Then we relate this notion to the notion of local comparison among abstractions (Cf. 7.4.3 on page 164).

**Definition 8.2.15 (local comparison of marker abstractions).** We say that the marker abstraction $(\mathscr{M}_1^\sharp, \mathscr{M}_2^\sharp, (\mathscr{M}_X^\sharp)_X)$ locally approximates the marker abstraction $(\overline{\mathscr{M}}_1^\sharp, \overline{\mathscr{M}}_2^\sharp, (\overline{\mathscr{M}}_X^\sharp)_X)$ if there exists some monotonic abstraction functions $\alpha^{\mathscr{M}_1^\sharp \leftarrow \overline{\mathscr{M}}_1^\sharp} \in \overline{\mathscr{M}}_1^\sharp \to \mathscr{M}_1^\sharp$, $\alpha^{\mathscr{M}_2^\sharp \leftarrow \overline{\mathscr{M}}_2^\sharp} \in \overline{\mathscr{M}}_2^\sharp \to \mathscr{M}_2^\sharp$, and $\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp} \in \overline{\mathscr{M}}_X^\sharp \to \mathscr{M}_X^\sharp$ for any finite set $X$, such that:

1. for any $i \in \{1;2\}$:

   (a) $\forall a \in \overline{\mathscr{M}}_i^\sharp,\ \gamma_i^{\overline{\mathscr{M}}}(a) \subseteq \gamma_i^{\mathscr{M}}(\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(a))$;

   (b) $\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(\overline{\bot}_i^{\mathscr{M}}) \sqsubseteq_i^{\mathscr{M}} \bot_i^{\mathscr{M}}$;

(c) $\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(\overline{\top}_i^{\mathscr{M}}) \sqsubseteq_i^{\mathscr{M}} \top_i^{\mathscr{M}}$;

(d) $\alpha^{\mathscr{M}_1^\sharp \leftarrow \overline{\mathscr{M}}_1^\sharp}(\overline{\varepsilon}_1^{\mathscr{M}}) \sqsubseteq_i^{\mathscr{M}} \varepsilon_1^{\mathscr{M}}$;

(e) $\forall A \in \wp_{\text{finite}}(\overline{\mathscr{M}}_i^\sharp)$,
$\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(\overline{\sqcup}_i^{\mathscr{M}}(A)) \sqsubseteq_i^{\mathscr{M}} \sqcup_i^{\mathscr{M}} (\{\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(a) \mid a \in A\})$;

(f) $\forall A \in \wp_{\text{finite}}(\overline{\mathscr{M}}_i^\sharp)$,
$\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(\overline{\sqcap}_i^{\mathscr{M}}(A)) \sqsubseteq_i^{\mathscr{M}} \sqcap_i^{\mathscr{M}} (\{\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(a) \mid a \in A\})$;

(g) $\forall a,b \in \overline{\mathscr{M}}_i^\sharp$,
$\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(a \overline{\nabla}_i^{\mathscr{M}} b) \sqsubseteq_i^{\mathscr{M}} (\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(a)) \nabla_i^{\mathscr{M}} (\alpha^{\mathscr{M}_i^\sharp \leftarrow \overline{\mathscr{M}}_i^\sharp}(b))$;

(h) $\forall a,b \in \overline{\mathscr{M}}_1^\sharp$,
$\alpha^{\mathscr{M}_2^\sharp \leftarrow \overline{\mathscr{M}}_2^\sharp}(\overline{\text{PAIR}}(a,b)) \sqsubseteq_2^{\mathscr{M}} \text{PAIR}(\alpha^{\mathscr{M}_1^\sharp \leftarrow \overline{\mathscr{M}}_1^\sharp}(a), \alpha^{\mathscr{M}_1^\sharp \leftarrow \overline{\mathscr{M}}_1^\sharp}(b))$;

(i) $\forall a \in \overline{\mathscr{M}}_2^\sharp$, $\alpha^{\mathscr{M}_2^\sharp \leftarrow \overline{\mathscr{M}}_2^\sharp}(\overline{\text{DIAG}}(a)) \sqsubseteq_2^{\mathscr{M}} \text{DIAG}(\alpha^{\mathscr{M}_2^\sharp \leftarrow \overline{\mathscr{M}}_2^\sharp}(a))$;

2. for any finite set $X$ and $Y$:

(a) $\forall a \in \overline{\mathscr{M}}_X^\sharp$, $\gamma_X^{\overline{\mathscr{M}}}(a) \subseteq \gamma_X^{\mathscr{M}}(\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a))$;

(b) $\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(\overline{\bot}_X^{\mathscr{M}}) \sqsubseteq_X^{\mathscr{M}} \bot_X^{\mathscr{M}}$;

(c) $\alpha^{\mathscr{M}_\emptyset^\sharp \leftarrow \overline{\mathscr{M}}_\emptyset^\sharp}(\overline{\emptyset^\sharp_{\mathscr{M}}}) \sqsubseteq_\emptyset^{\mathscr{M}} \emptyset^\sharp_{\mathscr{M}}$;

(d) $\forall A \in \wp_{\text{finite}}(\overline{\mathscr{M}}_X^\sharp)$,
$\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(\overline{\sqcup}_X^{\mathscr{M}} A) \sqsubseteq_X^{\mathscr{M}} \sqcup_X^{\mathscr{M}} (\{\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a) \mid a \in A\})$;

(e) $\forall q \in X \to Y, \forall a \in \overline{\mathscr{M}}_X^\sharp$,
$\alpha^{\mathscr{M}_Y^\sharp \leftarrow \overline{\mathscr{M}}_Y^\sharp}(\overline{\text{QUOTIENT}}(q,a)) \sqsubseteq_Y^{\mathscr{M}} \text{QUOTIENT}(q, \alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a))$;

(f) let $n \geq 2$ be an integer such that $\{(I,i) \mid 1 \leq i \leq n\} \subseteq X$, for any tuple $(p^i)_{1 \leq i \leq n}$ of program point labels, $\forall a \in \overline{\mathscr{M}}_X^\sharp$,
$\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(\underset{(p^i)}{\overline{\text{PUSH}}}(a)) \sqsubseteq_X^{\mathscr{M}} \underset{(p^i)}{\text{PUSH}}(\alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a))$;

(g) $\forall a \in \overline{\mathscr{M}}_X^\sharp, b \in \overline{\mathscr{M}}_Y^\sharp$,
$\alpha^{\mathscr{M}_{X\cup Y}^\sharp \leftarrow \overline{\mathscr{M}}_{X\cup Y}^\sharp}(a \overline{\otimes} b) \sqsubseteq_{X\cup Y}^{\mathscr{M}} \alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a) \otimes \alpha^{\mathscr{M}_Y^\sharp \leftarrow \overline{\mathscr{M}}_Y^\sharp}(b)$;

(h) $\forall a \in \overline{\mathscr{M}}_X^\sharp$,
$\alpha^{\mathscr{M}_{X\cap Y}^\sharp \leftarrow \overline{\mathscr{M}}_{X\cap Y}^\sharp}(\overline{\text{PROJ}}(X \cap Y, a)) \sqsubseteq_{X\cap Y}^{\mathscr{M}} \text{PROJ}(X \cap Y, \alpha^{\mathscr{M}_X^\sharp \leftarrow \overline{\mathscr{M}}_X^\sharp}(a))$;

3. let $X$ be a set of elements, let $x$ and $y$ be two distinct elements in $X$,

(a) $\forall a \in \overline{\mathscr{M}}^{\sharp}_{1}$,
$\alpha^{\mathscr{M}^{\sharp}_{\{x\}} \leftarrow \overline{\mathscr{M}}^{\sharp}_{\{x\}}}(\overline{\text{BUILD-SG}}_{x}(a)) \sqsubseteq^{\mathscr{M}}_{\{x\}} \text{BUILD-SG}_{x}(\alpha^{\mathscr{M}^{\sharp}_{1} \leftarrow \overline{\mathscr{M}}^{\sharp}_{1}}(a))$;

(b) $\forall a \in \overline{\mathscr{M}}^{\sharp}_{X}$,
$\alpha^{\mathscr{M}^{\sharp}_{1} \leftarrow \overline{\mathscr{M}}^{\sharp}_{1}}(\overline{\text{EXTRACT-SG}}_{x}(a)) \sqsubseteq^{\mathscr{M}}_{1} \text{EXTRACT-SG}_{x}(\alpha^{\mathscr{M}^{\sharp}_{X} \leftarrow \overline{\mathscr{M}}^{\sharp}_{X}}(a))$;

(c) $\forall a \in \overline{\mathscr{M}}^{\sharp}_{2}$,
$\alpha^{\mathscr{M}^{\sharp}_{\{x;y\}} \leftarrow \overline{\mathscr{M}}^{\sharp}_{\{x;y\}}}(\overline{\text{BUILD-PR}}_{x,y}(a)) \sqsubseteq^{\mathscr{M}}_{\{x;y\}} \text{BUILD-PR}_{x,y}(\alpha^{\mathscr{M}^{\sharp}_{2} \leftarrow \overline{\mathscr{M}}^{\sharp}_{2}}(a))$;

(d) $\forall a \in \overline{\mathscr{M}}^{\sharp}_{X}$,
$\alpha^{\mathscr{M}^{\sharp}_{2} \leftarrow \overline{\mathscr{M}}^{\sharp}_{2}}(\overline{\text{EXTRACT-PR}}_{x,y}(a)) \sqsubseteq^{\mathscr{M}}_{2} \text{EXTRACT-PR}_{x,y}(\alpha^{\mathscr{M}^{\sharp}_{X} \leftarrow \overline{\mathscr{M}}^{\sharp}_{X}}(a))$.

**Proposition 8.2.16.** *If the marker abstraction $(\mathscr{M}^{\sharp}_{1}, \mathscr{M}^{\sharp}_{2}, (\mathscr{M}^{\sharp}_{X})_{X})$ is locally approximating the marker abstraction $(\overline{\mathscr{M}}^{\sharp}_{1}, \overline{\mathscr{M}}^{\sharp}_{2}, (\overline{\mathscr{M}}^{\sharp}_{X})_{X})$, then the abstraction $\mathscr{A}$ is locally approximating the abstraction $\overline{\mathscr{A}}$.*

The following properties help in compositionally proving that a marker abstraction locally approximates an other marker abstraction:

**Proposition 8.2.17 (local comparison composition).** *The following properties are satisfied:*

1. *If the marker abstraction $\mathscr{O}$ locally approximates the marker abstraction $\mathscr{N}$ and if the marker abstraction $\mathscr{N}$ locally approximates the marker abstraction $\mathscr{M}$, then the marker abstraction $\mathscr{O}$ locally approximates the marker abstraction $\mathscr{M}$.*

2. *The marker abstraction $\mathscr{M}$ locally approximates the Cartesian product (see Sect. 8.2.3 on page 196) of the marker abstraction $\mathscr{M}$ and of the marker abstraction $\mathscr{N}$.*

3. *The marker abstraction $\mathscr{M}$ locally approximates any reduction (see Def. 8.2.4 on page 197) of the abstraction $\mathscr{M}$ by some anti-extensive operators.*

4. *If the marker abstraction $\mathscr{N}$ locally approximates the abstraction $\mathscr{O}$, then the Cartesian product (see Sect. 8.2.3 on page 196) of the marker abstractions $\mathscr{N}$ and $\mathscr{M}$ locally approximates the Cartesian product of the marker abstractions $\mathscr{O}$ and $\mathscr{M}$.*

5. *If:*

(a) *the marker abstraction $\mathcal{M}$ locally approximates the marker abstraction $\overline{\mathcal{M}}$, by using the abstraction functions $\alpha^{\mathcal{M}^\sharp_1 \leftarrow \overline{\mathcal{M}}^\sharp_1}$, $\alpha^{\mathcal{M}^\sharp_2 \leftarrow \overline{\mathcal{M}}^\sharp_2}$, and $(\alpha^{\mathcal{M}^\sharp_X \leftarrow \overline{\mathcal{M}}^\sharp_X})$,*

(b) *the marker abstractions $\mathcal{M}$ and $\overline{\mathcal{M}}$ are monotonic,*

(c) *$(\rho_1, \rho_2, (\rho_X))$ and $(\overline{\rho}_1, \overline{\rho}_2, (\overline{\rho}_X))$ are monotonic reduction operators such that:*

- *for any $x \in \overline{\mathcal{M}}_1$, $\alpha^{\mathcal{M}^\sharp_1 \leftarrow \overline{\mathcal{M}}^\sharp_1}(\overline{\rho}_1(x)) \sqsubseteq^{\mathcal{M}}_1 \rho_1(\alpha^{\mathcal{M}^\sharp_1 \leftarrow \overline{\mathcal{M}}^\sharp_1}(x))$,*
- *for any $x \in \overline{\mathcal{M}}_2$, $\alpha^{\mathcal{M}^\sharp_2 \leftarrow \overline{\mathcal{M}}^\sharp_2}(\overline{\rho}_2(x)) \sqsubseteq^{\mathcal{M}}_2 \rho_2(\alpha^{\mathcal{M}^\sharp_2 \leftarrow \overline{\mathcal{M}}^\sharp_2}(x))$,*
- *for any finite set $X$, for any $x \in \overline{\mathcal{M}}_X$, $\alpha^{\mathcal{M}^\sharp_X \leftarrow \overline{\mathcal{M}}^\sharp_X}(\overline{\rho}_X(x)) \sqsubseteq^{\mathcal{M}}_X \rho_X(\alpha^{\mathcal{M}^\sharp_X \leftarrow \overline{\mathcal{M}}^\sharp_X}(x))$;*

*then: the reduction of $\mathcal{M}$ by $(\rho_1, \rho_2, (\rho_X))$ locally approximates the reduction of $\overline{\mathcal{M}}$ by $(\overline{\rho}_1, \overline{\rho}_2, (\overline{\rho}_X))$.*

#### 8.2.7.2 Application

1. We denote by 0-CFA the uniform marker abstraction that is defined in Sect. 8.2.5.1 on page 198.

2. We denote by CONF the confinement marker abstraction that is defined in Sect. 8.2.5.2 on page 200.

3. We denote by $\text{LOC}_i$ the marker abstraction that is defined as the reduced product of:

   - the regular marker abstraction that is given in Sect. 8.2.5.3.1 on page 203
   - and the component-wise relational marker abstraction that is given in Sect. 8.2.5.3.2.1 on page 207,

   when each marker *id* is replaced with $\phi_i(id)$. We use the reduction operators that are defined in Sect. 8.2.5.3.3 on page 211.

4. We denote by $\text{GLOB}_i$ the marker abstraction that is defined as the reduced product of:

   - the regular marker abstraction that is given in Sect. 8.2.5.3.1 on page 203
   - and the global relational marker abstraction that is given in Sect. 8.2.5.3.2.2 on page 209,

when each marker *id* is replaced with $\phi_i(id)$. We use the reduction operators that are defined in Sect. 8.2.5.3.3 on page 211.

5. We denote by $\text{LOC}_i^{=}$ the marker abstraction that is defined as the reduced[9] product of $\text{LOC}_i$ and CONF.

6. We denote by $\text{GLOB}_i^{=}$ the marker abstraction that is defined as the reduced product of $\text{GLOB}_i$ and CONF.

We give in Fig. 8.4 a hierarchy of abstraction. All these abstractions are monotonic. Moreover, any edge from a marker abstraction $\mathscr{M}$ onto an other marker abstraction $\mathscr{N}$ means that the marker abstraction $\mathscr{N}$ locally approximates $\mathscr{M}$. Thus, the respective abstractions $\mathscr{A}_{\mathscr{M}}$ and $\mathscr{A}_{\mathscr{N}}$ satisfy $\gamma_{\mathscr{M}}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_{\mathscr{M}}}) \subseteq \gamma_{\mathscr{N}}(\llbracket \mathscr{S} \rrbracket_{\mathscr{A}_{\mathscr{N}}})$.

*Proof.* We prove the graph in Fig. 8.4.

1. The marker abstraction 0-CFA locally approximates the marker abstraction CONF (resp. $\text{LOC}_2$) by using the abstraction functions that maps any bottom element in graph (resp. non uniform) domains into the bottom element $\perp$ of the uniform domain and any other elements into the top element $\top$ of the uniform domain.

2. For any $i \in \{1;2\}$, the marker abstraction $\text{LOC}_i$ locally approximates the marker abstraction $\text{GLOB}_i$:

   - First, the component-wise relational abstraction locally approximates the global relational abstraction by using the abstraction functions that project each affine system $K$ over $\mathscr{V}_X$ for any finite set $X$ onto systems function $f$ that maps each letter $\lambda \in \Sigma_i$ to the projection of $K$ onto the set of variables $\{v_x^{\lambda} \mid x \in X\}$. This projection uses the Gaussian elimination.
   - Then, we use Prop. 8.2.17.(4) and Prop. 8.2.17.(5).

3. The marker abstraction $\text{LOC}_2$ locally approximates the marker abstraction $\text{LOC}_1$:

   - First, we prove that the regular abstraction that uses $\phi_2$ locally approximates the regular abstraction that uses $\phi_1$. The abstraction functions consists in quotienting graphs by the equivalence relation $\sim$ over $\mathscr{L}^{\geq 2}$, where $u \sim v$ if and only if $u_2 = v_2$.

[9] We use a monotonic and anti-extensive reduction that deduces equalities among markers from both equality among Parikh's vectors and assumptions about the shape of markers (see Sect. 8.3.3.2)

- Then, we prove that the component-wise relational abstraction that uses $\phi_2$ locally approximates the component-wise relational abstraction that uses $\phi_1$. The abstraction functions maps each $\lambda \in \mathscr{L}$ to the system that is obtained by:
  - gathering each system associated to any variable $\lambda' \in \mathscr{L}^{n \geq 2}$ such that $\lambda'_2 = \lambda$ where each occurrence of variable $v_x$ is replaced by the variable $v_x^{\lambda'}$;
  - inserting the constraints $v_x = \sum\{v_x^{\lambda'} \mid \lambda'_2 = \lambda\}$ for any $x$;
  - keeping only the constraints that involves the variables $v_x$ (by using Gaussian Elimination).
- We can conclude by applying Prop. 8.2.17.(4) twice and Prop. 8.2.17.(5).

4. The marker abstraction GLOB$_2$ locally approximates the marker abstraction GLOB$_1$: it is enough to prove that the global relational abstraction that uses $\phi_2$ locally approximates the global relational abstraction that uses $\phi_1$. The abstraction functions transforms each affine system $K$ over $\mathscr{V}_X$ as follows:

   (a) we introduce some extra variables $w_x^{\lambda}$ for any $x \in X$ and any $\lambda \in \mathscr{L}$;

   (b) we insert the constraint $w_x^{\lambda} = \sum_{u \in \mathscr{L}^{\geq 2},\, u_2 = \lambda} v_x^u$;

   (c) we project the system to keep the constraints that only involve the variables $w_x^{\lambda}$

   (d) we replace each occurrence of a variable $w_x^{\lambda}$ with the variable $x_x^{\lambda}$.

5. All other edges are proved by using Prop. 8.2.17.

□

## 8.3 More precise abstractions

The analyses proposed in Sect. 8.2 may only capture comparisons between thread markers and the values that are associated with the variables of its interface. In some cases, it turns out that there are no such relations whereas some relations between the values that are associated with several variables are useful.

**Example 8.3.1.** *We give in Fig. 8.5 a system written in the $\pi$-calculus. This system creates a communication ring between several monitors. The names of the channels opened by name restrictions* $(\nu$ **left0**$)$ *and* $(\nu$ ***right***$)$ *denote the processes of the ring. The first part of the system describes the ring creation. The first process*

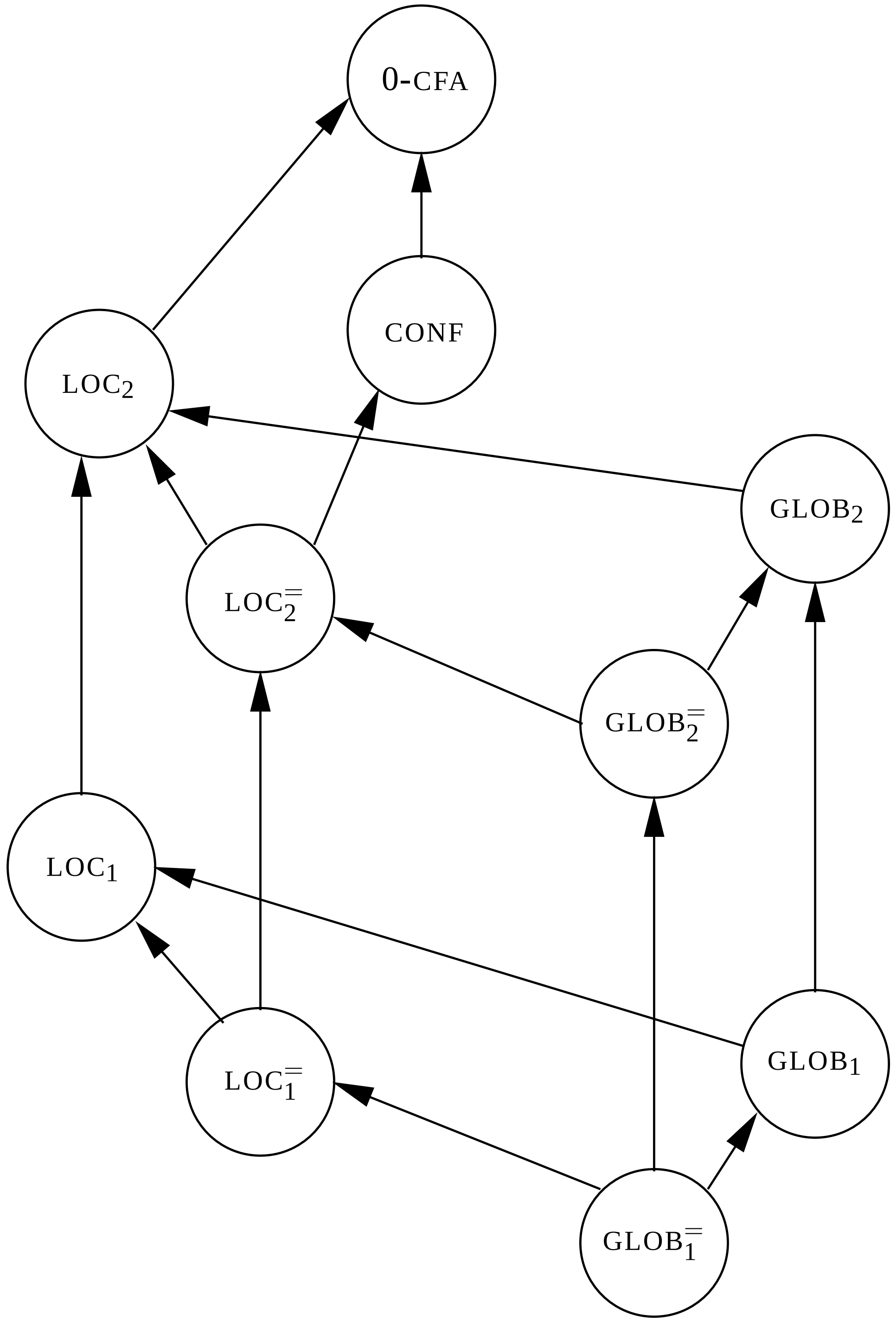


Figure 8.4: Abstraction hierarchy.

$(\nu \text{ ok})(\nu \text{ make})(\nu \text{ mon})(\nu \text{ left0})$
$($
$\quad(*\text{make?}^1[\mathit{left}](\nu\ \mathit{right})(\text{mon!}^2[\mathit{left},\mathit{right}] \mid \text{make!}^3[\mathit{right}]))$
$\mid (*\text{make?}^4[\mathit{left}](\text{mon!}^5[\mathit{left},\text{left0}]))$
$\mid \text{make!}^6[\text{left0}]$
$\mid \text{mon?}^7[x,y](\text{test!}^8[] \mid [x,y][x=y][x \neq \mathit{first}]\text{test?}^9[]\text{ok!}^{10}[])$
$)$

Figure 8.5: A ring of processes with a test.

*is created by the restriction* $(\nu$ **left0**$)$*. A thread* **mon**![***v1***;***v2***] *denotes a connection between two processes. Then, each time the first resource is replicated, a new process is created and linked to the previous process, which has been passed as an argument of the replication. The second resource replication closes the ring by linking the last created process to the first created process. The thread at program point* **7** *picks a connection between two monitors and checks whether the two monitors may be the same and whether the first argument is not the address of the first monitor of the ring. In such a case, it reaches the program point* **10***.*

*The analyses that we have described in Sect. 8.2 may not help in proving that the program point* **10** *is unreachable. The reason is that there is no relation at program point* **9** *between the marker of the thread (that is* $\varepsilon$*) and the value associated with the variable* **x***. This makes us miss the constraint between the values that are associated with the variables* **x** *and* **y***.* □

We propose in this section a wider class of abstract domains. These domains may especially express some relations between values, even if there is no relation between their markers and the marker of the thread instance they are communicated to. Nevertheless, this raises some complexity problems we propose to solve by designing several domains: there is a trade-off between information partitioning, and the accuracy of information propagation. Reduced product makes these domains collaborate. We also propose two particular domains that aim at discovering and propagating explicit equality and disequality relations among channel names and among markers.

## 8.3.1 Dependencies among thread names

### 8.3.1.1 Abstract domain of equality and disequality relations

We introduce an abstract domain for describing equality and disequality relations among a finite set of variables. We introduce for all finite set $X$ the abstract domain

$T_X$ of all the graphs $(P,E)$ such that $P$ is a partition of $X$ and $E$ is a part of $P \times P$.

Given abstract element $G = (P,E) \in T_X$, we introduce two binary relations over $X$ as follows:

- $a =_G b \overset{\Delta}{\iff} \exists C \in P, \{a,b\} \subseteq C$,
- $a \neq_G b \overset{\Delta}{\iff} \exists C_1, C_2 \in P,\ a \in C_1,\ b \in C_2, (C_1,C_2) \in E$;

we also define the function $[\_]_G \in X \to P$ that maps each element of $X$ into its equivalence class as $[x]_G = \{y \mid x =_G y\}$.

The domain $T_X$ is partially ordered by the $\leqslant_X^T$ relation defined as follows:

$$\forall G_1, G_2 \in T_X,\ G_1 \leqslant_X^T G_2 \overset{\Delta}{\iff} \begin{cases} \forall x,y \in X,\ x =_{G_2} y \Rightarrow x =_{G_1} y \\ \forall x,y \in X,\ x \neq_{G_2} y \Rightarrow x \neq_{G_1} y. \end{cases}$$

Roughly speaking, the partial order is the opposite of the constraint set inclusion because, the more constraints, the fewer solutions.

For every set $I$, each element $G$ in $T_X$ is related to the set of functions $\gamma_{T_X}^I$, defined as follows:

$$\gamma_{T_X}^I(G) = \left\{ f \in X \to I \;\middle|\; \forall x,\ y \in X,\ \begin{cases} x =_G y \implies f(x) = f(y) \\ x \neq_G y \implies f(x) \neq f(y) \end{cases} \right\}.$$

The concretization of a graph is the set of all the maps the image of which satisfy the equality and disequality relations that are described by this graph.

We also define some primitives over our abstract domain as follows:
Let $X$ and $Y$ be two finite sets:

1. we define an *abstract union*: gathering some abstract elements consists in intersecting the constraint sets they are described with: For any finite subset $A \in \wp(T_X) \setminus \emptyset$, we define $\sqcup_X^T A = (P', E')$ where the set of class $P'$ and the set of edges $E'$ are well-defined as follows:

$$\begin{cases} P' = \{\bigcap_{G \in A} [x]_G \mid x \in X\} \\ E' = \{(\bigcap_{G \in A} [x]_G, \bigcap_{G \in A} [y]_G) \in P'^2 \mid \forall G \in A,\ x \neq_G y\}; \end{cases}$$

The soundness of the abstract union is established by Prop. 8.3.2.

**Proposition 8.3.2.** *For any set $I$ and for any finite set $A \in \wp(T_X)$ and for any graph $G \in A$, we have $\gamma_{T_X}^I(G) \subseteq \gamma_{T_X}^I(\sqcup_X^T A)$.*

*Proof.* Let $X$ and $I$ be two sets. Let $A \in \wp(T_X)$. Let $G = (P,E) \in A$. We denote $G_\sqcup = \sqcup_X^T A = (P_\sqcup, E_\sqcup)$. Let $f \in X \to I$ be a function such that $f \in \gamma_{T_X}^I(G)$.

Let $x$, $y$ be two elements in $X$.

(a) In the case when $x =_{G_\sqcup} y$: we have $y \in [x]_{G_\sqcup}$, so $y \in [x]_{G'}$ for any $G' \in A$. Since $G \in A$, we have $y \in [x]_G$. We conclude that $x =_G y$. By definition of $\gamma^I_{T_X}(G)$, we get that $f(x) = f(y)$.

(b) In the case when $x \neq_{G_\sqcup} y$: we have $([x]_{G_\sqcup}, [y]_{G_\sqcup}) \in E'$, there exists two elements $x' \in [x]_{G_\sqcup}$ and $y' \in [y]_{G_\sqcup}$ such that $x' \neq_{G'} y'$ for any $G' \in A$. By applying the previous case, we have $f(x) = f(x')$ and $f(y) = f(y')$. Since $G \in A$, we have $x' \neq_G y'$. By definition of $\gamma^I_{T_X}(G)$, we get that $f(x') \neq f(y')$. We conclude that $f(x) \neq f(y)$.

By definition, we have $f \in \gamma^I_{T_X} G_\sqcup$. □

2. we define an *abstract join* operator: for any graph $G_X \in T_X$, $G_Y \in T_Y$, we define $G_X \otimes^T G_Y \in T_{X\cup Y}$ as $(P', E')$ where the set of classes $P'$ and the set of edges $E'$ are defined as follows:

   - $P' = \{lfp[A \to f(v,A)] \mid v \in X \cup Y\}$, where $f(v,A)$ is given by

$$\{v\} \cup \left\{ t \in X \cup Y \;\middle|\; \begin{array}{l} \exists u \in A,\ [(t,u) \in X^2 \setminus Y^2 \text{ and } t =_X u] \\ \text{or } [(t,u) \in Y^2 \setminus X^2 \text{ and } t =_Y u] \\ \text{or } [(t,u) \in X^2 \cap Y^2, t =_X u, \text{ and } t =_Y u] \end{array} \right\}.$$

   - $E' = \left\{ (C_1, C_2) \in (P')^2 \;\middle|\; \begin{array}{l} \exists (t,u) \in (C_1, C_2), \\ [(t,u) \in X^2 \setminus Y^2 \text{ and } t \neq_X u] \\ \text{or } [(t,u) \in Y^2 \setminus X^2 \text{ and } t \neq_Y u] \\ \text{or } [(t,u) \in X^2 \cap Y^2, t \neq_X u, \text{ and } t \neq_Y u \end{array} \right\}.$

   Roughly speaking, we partition the set $X \cup Y$ into three parts $X \setminus Y$, $Y \setminus X$ and $X \cap Y$. A relation that constrains an element in $X \setminus Y$ (resp. in $Y \setminus X$) is always kept, while a relation between two elements in $X \cap Y$ is only kept providing that it occurs both in $G_X$ and in $G_Y$. Then the abstract join operator computes the closure of these constraints. The soundness of the abstract join operator is established by Prop. 8.3.3.

   **Proposition 8.3.3.** *For any set I and for any graph $G_X \in T_X$, $G_Y \in T_Y$, we have:*

$$\left\{ f \in (X \cup Y) \to I \;\middle|\; \begin{cases} f_{|X} \in \gamma^I_{T_X}(G_X) \\ f_{|Y} \in \gamma^I_{T_X}(G_Y) \end{cases} \right\} \subseteq \gamma^I_{T_{X\cup Y}}(G_X \otimes^T G_Y).$$

3. we define an *abstract projection*: It just consists in restricting the set of vertice, keeping equality and disequality relations on the remaining vertice.

for any $G = (P,E) \in T_X$, we define $\text{PROJ}_Y^T(G) \in T_Y$ as $(P',E')$ where the set of classes $P'$ and the set of edges $E'$ are defined as follows:

$$\begin{cases} P' = \{C \cap Y \mid C \in P,\ C \cap Y \neq \emptyset\}, \\ E' = \{(C_1 \cap Y, C_2 \cap Y) \mid (C_1,C_2) \in E \cap (P')^2\}. \end{cases}$$

The soundness of the abstract projection is established by Prop. 8.3.4.

**Proposition 8.3.4.** *For any set $I$ and for any graph $G \in T_X$, we have:*

$$\left\{ f_{|X\cap Y} \;\middle|\; f \in \gamma^I_{T_X}(G) \right\} \subseteq \gamma^I_{T_{X\cap Y}}(\text{PROJ}_Y^T G).$$

4. *renaming*: for any bijection $g \in X \to Y$ and any $G = (P,E) \in T_X$ we define the renaming of $G$ by $g$, $\text{RENAME}_g(G)$, as $(P',E')$ where:

$$\begin{cases} P' = \{g(C) \mid C \in P\}, \\ E' = \{(g(C_1), g(C_2)) \mid (C_1,C_2) \in E\}. \end{cases}$$

The soundness of the abstract renaming is established by Prop. 8.3.5.

**Proposition 8.3.5.** *For any set $I$, for any graph $G_X \in T_X$ and for any bijection $g \in X \to Y$, we have:*

$$\{f \mid f \circ g \in \gamma^I_{T_X}(G_X)\} \subseteq \gamma^I_{T_Y}(\text{RENAME}_g(G_X)).$$

Then we define an emptiness test $\text{EMPTY}_X \in \wp(T_X)$ as $G = (P,E) \in \text{EMPTY}_X \iff \exists C \in P,\ (C,C) \in E$. This emptiness test satisfies the following soundness properties:

**Proposition 8.3.6.** *For any finite set $I$ and for any $G \in \text{EMPTY}_X$, we have $\gamma^I_{T_X}(G) = \emptyset$.*

### 8.3.1.2 Equality and disequality relations among channel names

First we abstract the equality and the disequality relations between the values of each thread.

**8.3.1.2.1 Atom abstraction** For each interface $V$, we define the domain $Atome_V^{=1} = T_V$. Each abstract element $G \in Atome_V^{=1}$ is related to the set $\gamma_V^{=1}(G) = \mathscr{M} \times \gamma_{T_V}^{\mathscr{L}\times\mathscr{M}}(G)$. The structure of the domains $Atome_V^{=1}$ is given as follows: we set $\sqsubseteq_V^{=1} = \leqslant_V^T$, $\sqcup_V^{=1} = \sqcup_V^T$, $\bot_V^{=1} = (V, V \times V)$. Since there is no infinite increasing sequences in $T_V$, we define $a \nabla_V^{=1} b = \sqcup_V^{=1}\{a,b\}$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:

$$\varepsilon^{=1} = (\emptyset, \emptyset).$$

- the abstract restriction is defined as follows:

$$\nu^{=1}(x, l, (P, E)) = (P \cup \{x\}, E \cup (\{x\} \times E)).$$

- the abstract garbage collection is defined as follows:

$$\text{GC}^{=1}(X, G) = \text{PROJ}_X^T(G).$$

**Theorem 8.3.7.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

**8.3.1.2.2 Molecule abstraction** For any finite $n$-tuple $(V_i)_{1 \leq i \leq n}$ of interfaces. we define the domain $\mathit{Molecule}^{=1}_{(V_i)_{1 \leq i \leq n}}$ as the domain $T_{\{(X,k) \mid X \in V_k\}}$. Each abstract element $G \in \mathit{Molecule}^{=1}_{(V_i)}$ is related to the set $\gamma^{=1}_{(V_i)}(G)$ of families $(id_i, E_i) \in \Pi_{1 \leq i \leq n} \mathit{Env}^{\mathscr{M}}_{V_i}$ of pair marker/environment such that the function $g \in \{(X,k) \mid X \in V_k\} \rightarrow \mathscr{M}$ that is defined by $g(X,k) = E_k(X)$ is in the set $\gamma_{T_{\mathscr{M}}}^{\{(X,k) \mid X \in V_k\}}(G)$.

Abstract primitives are defined as follows:

- the abstract injection just renames variables: we define $\text{INJ}^{=1}(A) = \text{RENAME}_{[X \mapsto (X,1)]}(A)$.
- the abstract concatenation is computed by re-indexing the variables of the right arguments and by joining the two graphs. Let $m, n$ be two integers. Let $(U_i)_{1 \leq i \leq m}$ and $(V_i)_{1 \leq i \leq n}$ be two families of interfaces. Let $G_1 \in \mathit{Molecule}^{=1}_{(U_i)}$ and $G_2 \in \mathit{Molecule}^{=1}_{(V_i)}$ be two abstract molecules. We define the abstract product $G_1 \bullet^{=1} G_2$ of $G_1$ and $G_2$ as $G_1 \otimes^T \text{RENAME}_{[(X,k) \mapsto (X,k+m)]}(G_2)$.
- first the abstract projection isolates the information about one thread, and then renames the related variables. Let $n$ be an integer and $(V_i)_{1 \leq i \leq n}$ be a family of interfaces. Let $k$ be an integer such that $1 \leq k \leq n$. We set $\text{PROJ}^{=1}(k, G) = \text{RENAME}_{[(X,k) \mapsto X]}(\text{PROJ}^T_{\{(X,k) \mid X \in V_k\}}(G))$.
- first the abstract extension projects the constraints to keep only information about the variables that are not redefined and then by inserting redefined variables without any constraint about them. Let $n$ be an integer and $(V_i)_{1 \leq i \leq n}$ be a family of interfaces. Let $X$ be a subset of $\mathscr{V} \times [\![1; n]\!]$. Let $G$ be an element of $\mathit{Molecule}^{=1}_{(V_i)}$. First we compute $(P', E') = \text{PROJ}^T_{\{(x,i) \mid x \in V_i\} \setminus X}(G)$. We then set $\text{NEW}^{=1}_{\top}(X, G) = (P' \cup X, E')$.

- the abstract synchronization of an abstract element $G = (P, E)$ with respect to a constraint set $S$ is given by first quotienting the partition $P$ according to the equality constraints in $S$ and by then updating the set of edges according to the disequality constraints in $S$.

  Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program points. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces. Let $G = (P, E) \in Molecule^{\overline{=}1}_{(V_i)}$ be an abstract molecule. Let $S$ be a set of constraints $(x,k) \diamond (y,l)$ with $1 \leq k \leq n$, $x \in V_k \cup \{I\}$, $1 \leq l \leq n$, and $y \in V_l \cup \{I\}$. We introduce the binary equivalence $=_S$ over $P$ that is defined by $C =_S C'$ if there exist two variables $a \in C$ and $a' \in C'$, an integer $m \in \mathbb{N}$, and a sequence $a_0, \ldots, a_m$ of variables in $\{(x,k) \mid 1 \leq k \leq n,\ x \in V_k \cup \{I\}\}$ such that $(a_i = a_{i+1}) \in S$ or $(a_{i+1} = a_i) \in S$, for any $i$ such that $0 \leq i < m$, and such that $a = a_0$ and $a' = a_m$. For any class $C \in P$, we denote $C_{=_S}$ the set $\bigcup\{D \in P \mid C =_S D\}$. Then we define $(P', E')$ as:

  - $P' = \{C_{=_S} \mid C \in P\}$;
  - $E' = \{(X_{=_S}, Y_{=_S}) \mid (X,Y) \in E\} \cup \{(X,Y) \in (P')^2 \mid \exists x \in X, y \in Y,\ (x \neq y) \in S\}$.

  Then we distinguish two cases when defining the abstract synchronization:

  1. in the case when $(P', E') \in \text{EMPTY}_{\{(X,k) \mid 1\leq k\leq n,\ X\in V_k\}}$, we set $\text{SYNC}^{\overline{=}1}(S, (p_i), G) = (U, U \times U)$ where $U = \{(X,k) \mid 1 \leq k \leq n,\ X \in V_k\}$.
  2. otherwise, we set $\text{SYNC}^{\overline{=}1}(S, G) = (P', E')$.

- marker allocation does not change values, so we set $\text{FETCH}^{\overline{=}1}((p_i), G) = G$.

**Theorem 8.3.8.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

#### 8.3.1.3 Equality and disequality relations among markers

Then we abstract the equality and the disequality relations between the marker of a thread and the markers of its names.

**8.3.1.3.1 Atom abstraction** For each interface $V$, we define the domain $Atome^{\overline{=}2}_V = T_{V\cup\{I\}}$. Each abstract element $G \in Atome^{\overline{=}2}_V$ is related to the set $\gamma^{\overline{=}2}_V(G) = \{(id, E) \mid [I \mapsto id, X \to snd(E(X))] \in \gamma^{\mathscr{M}}_{T_{V\cup\{I\}}}(G)$. The structure of the domains $Atome^{\overline{=}2}_V$ is given as follows: we set $\sqsubseteq^{\overline{=}2}_V = \leqslant^T_{V\cup\{I\}}$, $\sqcup^{\overline{=}2}_V = \sqcup^T_{V\cup\{I\}}$, $\perp^{\overline{=}2}_V = (V \cup \{I\}, (V \cup \{I\})^2)$. Since there is no infinite increasing sequences in $T_{V\cup\{I\}}$, we define $a \nabla^{\overline{=}2}_V b = \sqcup^{\overline{=}2}_V \{a, b\}$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:
$$\varepsilon^{=_2} = (\{I\}, \emptyset).$$
- the abstract restriction uses the fact that the marker of fresh values is necessarily the marker of the threads that declare these values. Let $V \subseteq \mathscr{V}$ be a finite set of variables. Let $x \in \mathscr{V}$ be a variable. Let $l \in \mathscr{L}$ be a label. Let $(P,E) \in Atome_V^{=_2}$ be an abstract element. First we define the function $\sigma$:
$$\sigma(C) = \begin{cases} C & \text{if } I \notin C \\ C \cup \{x\} & \text{otherwise} \end{cases}$$
which inserts the variable $x$ into the equivalence class of the identity $I$ of the thread. Then the abstract restriction is defined as follows:
$$\nu^{=_2}(x,l,(P,E)) = (\{\sigma(C) \mid C \in P\}, \{(\sigma(C_1),\sigma(C_2)) \mid (C_1,C_2) \in E\})$$
by applying the function $\sigma$ to each equivalence class.
- the abstract garbage collection is defined as follows:
$$\text{GC}^{=_2}(X,G) = \text{PROJ}^{T}_{X \cup \{I\}}(G).$$

**Theorem 8.3.9.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

**8.3.1.3.2 Molecule abstraction** For any finite $n$-tuple $(V_i)_{1 \le i \le n}$ of interfaces. we define the domain $Molecule^{=_2}_{(V_i)_{1\le i\le n}}$ as the domain $T_{\{(X,k) \mid X \in V_k \cup \{I\}\}}$. Each abstract element $G \in Molecule^{=_2}_{(V_i)}$ is related to the set $\gamma^{=_2}_{(V_i)}(G)$ of families $(id_i, E_i) \in \Pi_{1 \le i \le n} Env^{\mathscr{M}}_{V_i}$ of pairs marker/environment such that the function $g \in \{(X,k) \mid X \in V_k \cup \{I\}\} \to \mathscr{M}$ that is defined by $g(X,k) = E_k(X)$ when $X \in V_k$ and by $g(I,k) = id_k$ is in the concretization $\gamma^{\{(X,k) \mid X \in V_k\}}_{T_{\mathscr{M}}}(G)$.

Abstract primitives are defined as follows:

- the abstract injection just renames variables: we define $\text{INJ}^{=_2}(A) = \text{RENAME}_{[X \mapsto (X,1)]}(A)$.
- the abstract concatenation is computed by re-indexing the variables of the right arguments and by joining the two graphs. Let $m,n$ be two integers. Let $(U_i)_{1 \le i \le m}$ and $(V_i)_{1 \le i \le n}$ be two families of interfaces. Let $G_1 \in Molecule^{=_2}_{(U_i)}$ and $G_2 \in Molecule^{=_2}_{(V_i)}$ be two abstract molecules. We define the abstract product $G_1 \bullet^{=_2} G_2$ of $G_1$ and $G_2$ as $G_1 \otimes^{T} \text{RENAME}_{[(X,k) \mapsto (X,k+m)]}(G_2)$.

- first the abstract projection isolates the information about one thread, and then it renames the related variables. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $k$ be an integer such that $1 \leq k \leq n$. We set $\text{PROJ}^{=_2}(k,G) = \text{RENAME}_{[(X,k)\mapsto X]}(\text{PROJ}^{T}_{\{(X,k)\ \mid\ X\in V_k\cup\{I\}\}}(G))$.

- first the abstract extension projects the constraints to keep only information about the variables that are not redefined and then it adds redefined variables without any constraint about them. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $X$ be a subset of $\mathscr{V} \times [\![1;n]\!]$. Let $G$ be an element of $Molecule^{=_2}_{(V_i)}$. First we compute $(P',E') = \text{PROJ}^{T}_{\{(x,i)\ \mid\ x\in V_i\cup\{I\}\}\setminus X}(G)$. Then we set $\text{NEW}^{=_2}_{\top}(X,G) = (P' \cup X, E')$.

- the abstract synchronization of an abstract element $G = (P,E)$ with respect to a constraint set $S$ is given by first quotienting the partition $P$ according to the equality constraints in $S$; disequality constraints do not give any information.

  Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program point. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces. Let $G = (P,E) \in Molecule^{=_1}_{(V_i)}$ of $(P,E)$. Let $S$ be a set of constraints $(x,k) \diamond (y,l)$ with $1 \leq k \leq n$, $x \in V_k \cup \{I\}$, $1 \leq l \leq n$, and $y \in V_l \cup \{I\}$. We introduce the binary equivalence $=_S$ over $P$ that is defined by $C =_S C'$ if there exist two variables $a \in C$ and $a' \in C'$, an integer $m \in \mathbb{N}$, and a sequence $a_0,\ldots,a_m$ of variables in $\{(x,k) \mid x \in V_k \cup \{I\}\}$ such that $a_i = a_{i+1} \in S$ or $a_{i+1} = a_i \in S$, for any k such that $0 \leq i < m$, and such that $a_0 = a$ and $a_m = a'$. For any class $C \in P$, we denote $C_{=_S}$ the set $\bigcup\{D \in P \mid C =_S D\}$. Then we define $\text{SYNC}^{=_2}(S,(p_i),G)$ as $(P',E')$ where:

  - $P' = \{C_{=_S} \mid C \in P\}$;
  - $E' = \{(X_{=_S}, Y_{=_S}) \mid (X,Y) \in E\}$.

- the abstract marker allocation consists in taking into account the fact the the allocated marker is fresh. First we forget any constraints about the identity of the first thread; then we insert the constraints that the new marker is distinct from any other marker. We first define the function $\sigma$ that removes the variable $(I,1)$ from any equivalence class:

  $$\sigma(C) = C \setminus \{(I,1)\}.$$

  Then, we define $\text{FETCH}^{=_2}((p_i),G)$ as $(P',E')$ where:

  $$\begin{cases} P' = \{(I,1)\} \cup (\{\sigma(C) \mid C \in P\} \setminus \{\emptyset\}), \\ E' = \left\{ (\sigma(C_1),\sigma(C_2)) \;\middle|\; \begin{array}{l} (C_1,C_2) \in E, \\ C_1,C_2 \in P \setminus \{(I,1)\} \end{array} \right\} \cup \{(\{(I,1)\},C) \mid C \in P'\}. \end{cases}$$

**Theorem 8.3.10.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

## 8.3.2 Marker analysis

In this section we abstract the shape of the markers that occur in the system during any computation sequences. We are both interested in the markers that may be associated with each thread and in the markers that may be associated with each variable in a thread interface. For the sake of simplicity, we use Prop. 4.5.18 on page 69 and approximate every tree marker $id$ by the word $\phi_2(id)$ of $(\mathscr{L})^*$.

First we describe the general shape of markers, and then we infer some relational algebraic properties on them. By reduction, we will use this information to synthesize equality and disequality relations between variable values and between markers.

### 8.3.2.1 Shape analysis

Shape analysis consists in distinctly abstracting, for each program point $p$, the set of markers which may be associated to a thread at program point $p$ and the set of values that may be associated with each variable $v$ in the interface $I(p)$ of threads at program point $p$. For the sake of efficiency, we do not partition abstract information according to the label $l$ of each potential value $(l, id)$: we abstract each value $(l, id)$ by the word $l.\phi_2(id)$ in the language $(\mathscr{L})^*$.

**8.3.2.1.1 Domain** We are left to abstract sets of words. For that purpose we will use the abstract domain $\text{Reg}_{\mathscr{L}}$ that we have defined in Sect. 8.2.5.3.1 on page 203. In order to build the abstraction of a value when knowing both the label $l$ and the marker abstraction $(i, f, t, b) \in \text{Reg}_{\mathscr{L}}$, we use another primitive $\text{PREFIX}_{\mathscr{L}}^{\text{Reg}}$. Then the abstract element $\text{PREFIX}_{\mathscr{L}}^{\text{Reg}}(l, (i, f, t, b))$ is defined as:

$$(\{l\}, f \cup \{l \mid b = 1\}, t[l \mapsto t(l) \cup i], 0).$$

Moreover we have:

$$\{l.id \mid id \in \gamma_{\mathscr{L}}^{\text{Reg}}((i, f, t, b))\} \subseteq \gamma_{\mathscr{L}}^{\text{Reg}}(\text{PREFIX}_{\mathscr{L}}^{\text{Reg}}(l, (i, f, t, b))).$$

Conversely we introduce the primitive $\text{TAIL}_{\mathscr{L}}^{\text{Reg}}$ that removes the first letter of a word. We set $\text{TAIL}_{\mathscr{L}}^{\text{Reg}}((i, f, t, b)) = (i', f', t', b')$ where:

- $i' = \bigcup_{a \in i} t(a)$;
- $f' = f$;

- $t' = t$;
- $b' = \begin{cases} 1 & \text{if } i \cap f \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$

We have:

$$\{id \mid \exists a \in \mathscr{L},\ a.id \in \gamma_{\mathscr{L}}^{\text{Reg}}((i,f,t,b))\} \subseteq \gamma_{\mathscr{L}}^{\text{Reg}}(\text{TAIL}_{\mathscr{L}}^{\text{Reg}}((i,f,t,b))).$$

**8.3.2.1.2 Atom abstraction** For each interface $V$, we define the domain $Atome_V^{\text{SHAPE}} = (V \cup \{I\} \to \text{Reg}_{\mathscr{L}})$. Each abstract element $f \in Atome_V^{\text{SHAPE}}$ is related to the set:

$$\gamma_V^{\text{SHAPE}}(f) = \gamma_{\mathscr{L}}^{\text{Reg}}(f(I)) \times \Pi_{v \in V}\{(l,id) \mid l.id \in \gamma_{\mathscr{L}}^{\text{Reg}}(f(v))\}.$$

The structure $(\sqsubseteq_V^{\text{SHAPE}}, \sqcup_V^{\text{SHAPE}}, \bot_V^{\text{SHAPE}})$ of the domain $Atome_V^{\text{SHAPE}}$ is defined component-wise from the domain structure $(\sqsubseteq_{\mathscr{L}}^{\text{Reg}}, \sqcup_{\mathscr{L}}^{\text{Reg}}, \bot_{\mathscr{L}}^{\text{Reg}})$. Since there is no infinite increasing sequences in $Atome_V^{\text{SHAPE}}$, we define $a \nabla_V^{\text{SHAPE}} b = \sqcup_V^{\text{SHAPE}}\{a,b\}$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:

$$\varepsilon^{\text{SHAPE}} = [I \mapsto \text{EMPTY}_{\mathscr{L}}^{\text{Reg}}].$$

- the abstract restriction is defined as follows:

$$\nu^{\text{SHAPE}}(x,l,f) = f[x \mapsto \text{PREFIX}_{\mathscr{L}}^{\text{Reg}}(l, f(I))].$$

- the abstract garbage collection is defined as follows:

$$\text{GC}^{\text{SHAPE}}(X,f) = f_{|(V \cap X) \cup \{I\}}.$$

**Theorem 8.3.11.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

**8.3.2.1.3 Molecule abstraction** For any finite $n$-tuple $(V_i)_{1 \leq i \leq n}$ of interfaces, we define the domain $Molecule_{(V_i)_{1 \leq i \leq n}}^{\text{SHAPE}}$ as the domain $\{(X,i) \mid 1 \leq i \leq n,\ X \in V_i \cup \{I\}\} \to \text{Reg}_{\mathscr{L}}$. Each abstract element $f \in Molecule_{(V_i)}^{\text{SHAPE}}$ is related to the set $\gamma_{(V_i)}^{\text{SHAPE}}(f)$ of the families $(id_i, E_i) \in \Pi_{1 \leq i \leq n} Env_{V_i}^{\mathscr{M}}$ of pair marker/environment such that for any integer $i$ such that $1 \leq i \leq n$, we have $id_i \in \gamma_{\mathscr{L}}^{\text{Reg}}(f((I,i)))$ and $[E_i(v) = (l, id_l) \implies l.id_l \in \gamma_{\mathscr{L}}^{\text{Reg}}(f((v,i)))$ for any $v \in V_i$.

Abstract primitives are defined as follows:

- the abstract injection just renames variables: we define $\text{INJ}^{\text{SHAPE}}(f) = [(X,1) \mapsto f(X)]$.

- the abstract concatenation is computed by re-indexing the variables of the right arguments and by joining the two maps. Let $m,n$ be two integers. Let $(U_i)_{1\leq i\leq m}$ and $(V_i)_{1\leq i\leq n}$ be two families of interfaces. Let $f_1 \in \mathit{Molecule}^{\text{SHAPE}}_{(U_i)}$ and $f_2 \in \mathit{Molecule}^{\text{SHAPE}}_{(V_i)}$ be two abstract molecules. We define the abstract product $f_1 \bullet^{\text{SHAPE}} f_2$ of $f_1$ and $f_2$ as the following function:
$$\begin{cases} (X,i) \mapsto f_1(X,i) & \text{if } 1 \leq i \leq m \\ (X,i) \mapsto f_2(X,i-m) & \text{if } m+1 \leq i \leq n. \end{cases}$$

- the abstract projection just keeps information about a given thread and renames the corresponding variables. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $k$ be an integer such that $1 \leq k \leq n$. We set $\text{PROJ}^{\text{SHAPE}}(k,f) = [v \in V_k \cup \{I\} \mapsto f((v,k))]$.

- the abstract extension removes any constraint about the redefined variables. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $X$ be a subset of $\mathscr{V} \times [\![1;n]\!]$. Let $f$ be an element of $\mathit{Molecule}^{\text{SHAPE}}_{(V_i)}$. Then we set $\text{NEW}^{\text{SHAPE}}_{\top}(X,f) = f[v \in X \mapsto \top^{\text{Reg}}_{\mathscr{L}}]$.

- the abstract synchronization of an abstract element $f$ with respect to a constraint set $S$ consists in taking the meet of the information about equal markers (we make the transitive closure of equality constraints to get more information). Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program point. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces. Let $S$ be a set of constraints $(x,k)\diamond(y,l)$ with $1 \leq k \leq n$, $x \in V_k \cup \{I\}$, $1 \leq l \leq n$, and $y \in V_l \cup \{I\}$. We introduce the binary equivalence $=_S$ over $\mathscr{V}$ that is defined by $(x,k) =_S (y,l)$ if there exists an integer $m \in \mathbb{N}$ and a sequence $a_0,\ldots,a_m$ of variables in $\{(x,k) \mid x \in V_k \cup \{I\}\}$ such that $a_i = a_{i+1} \in S$ or $a_{i+1} = a_i \in S$, for any $i$ such that $0 \leq i < m$ and such that $(x,k) = a_0$ and $(y,l) = a_m$. For any variable $X = (x,k)$, we denote $X_{=_S}$ the set $\bigcup\{Y \mid X =_S Y\}$. The value abstraction of a variable $(x,k)$ (where $1 \leq k \leq n$ and $x \in V_k \cup \{I\}$) before having taken into account synchronization constraints is denoted by $g((p_i),x,k)$. It is defined as follows:
$$g((p_i),x,k) \triangleq \begin{cases} \text{PREFIX}^{\text{Reg}}_{\mathscr{L}}(p_k, f(x,k)) & \text{if } x = I, \\ f(x,k) & \text{otherwise.} \end{cases}$$

  The value abstraction of a variable $(x,k)$ (where $1 \leq k \leq n$ and $x \in V_k \cup \{I\}$) after having taking into account synchronization constraints $S$ is denoted by

$h((p_i),S,x,k)$. It is defined as follows:

$$h((p_i),S,x,k) \triangleq \sqcap^{\mathrm{Reg}}_{\mathscr{L}}\{g((p_i),y,l) \mid (y,l) \in (x,k)_{=S}\}.$$

We can now define $\textsc{sync}^{\textsc{shape}}(S,(p_i),f)$ as follows:

$$\textsc{sync}^{\textsc{shape}}(S,(p_i),f)(x,k) \triangleq \begin{cases} \bot^{\mathrm{Reg}}_{\mathscr{L}} & \text{if } \exists(y,l),\ g(y,l) = \bot^{\mathrm{Reg}}_{\mathscr{L}}, \\ h((p_i),S,x,k) & \text{otherwise.} \end{cases}$$

- marker allocation updates the abstraction of the first thread marker, by using the abstraction of the second thread marker. We set:

$$\textsc{fetch}^{\textsc{shape}}((p^i),f) \triangleq f[(I,1) \mapsto \textsc{push}^{\mathrm{Reg}}_{\mathscr{L}}(p^2, f(I,2))].$$

**Theorem 8.3.12.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

#### 8.3.2.2 Global numerical abstraction

Numerical abstraction captures the relations between the markers which are associated to each thread and to their values. This abstraction is built upon the lattice of affine relations among a set of numerical variables [47]. First each word is approximated by its Parikh's vector [64], then we abstract the relations between occurrence numbers of letters in markers.

Unlike the domains that we have described in Sect. 8.2.5.3.2, we abstract sets of pairs marker/environment globally, without partitioning the abstraction according to the label of the values. This way, a pair marker/environment is seen as a huge vector. Then the set of such vectors are approximated by its affine hull. We take into the fact that a label $l$ is the label of the value of a variable $v$ by encoding it in some extra variables, instead of using partitioning.

**8.3.2.2.1 Atom abstraction** For each $V \subseteq \mathscr{V}$, we denote by $\mathscr{X}_V$ the set of variables $\{\mathrm{P}_\lambda \mid \lambda \in \mathscr{L}\} \cup \{\mathrm{V}_{(\lambda,v)} \mid \lambda \in \mathscr{L},\ v \in V\} \cup \{\mathrm{B}_{(l,v)} \mid l \in \mathscr{L},\ v \in V\}$. The variable $\mathrm{P}_\lambda$ describes the number of occurrences of $\lambda$ in the thread marker (or in the abstraction of the marker), while the variable $\mathrm{V}_{(\lambda,v)}$ is used to count the number of occurrences of $\lambda$ in the marker of the value of the variable $v$. The variable $\mathrm{B}_{(l,v)}$ encodes whether the label $l$ is the label of the value of the variable $v$, or not. The domain $Atome^{\mathrm{GLOB}}_V$ is the set of affine equality relations among the variables of $\mathscr{X}_V$. For all affine system $K$ in $Atome^{\mathrm{GLOB}}_V$, $\gamma^{\mathrm{GLOB}}_V(K) \subseteq Env^{\mathscr{M}}_V$ is the set of the elements $(id, E)$ such that the assignment[10]:

$$\left[\begin{array}{lcl} \mathrm{P}_\lambda & \to & |\phi_2(id)|_\lambda, \\ \mathrm{V}_{(\lambda,v)} & \to & |\phi_2(id_v)|_\lambda \text{ where } (l', id_v) = E(v)], \\ \mathrm{B}_{(l,v)} & \to & \delta_l^{l'} \text{ where } (l', id_v) = E(v)] \end{array}\right]$$

is a solution of $K$.

The structure $(\sqsubseteq_V^{\mathrm{GLOB}}, \sqcup_V^{\mathrm{GLOB}}, \bot_V^{\mathrm{GLOB}})$ is defined using affine operators described in [47]. Since there is no increasing infinite sequences in $Atome_V^{\mathrm{GLOB}}$, we define $a\nabla_V^{\mathrm{GLOB}} b = \sqcup_V^{\mathrm{GLOB}}\{a, b\}$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:

$$\varepsilon^{\mathrm{GLOB}} = \{\mathrm{P}_\lambda = 0,\ \forall \lambda \in \mathscr{L}\ .$$

- the abstract restriction $\nu^{\mathrm{GLOB}}(x, l, K)$ is obtained in inserting in the affine system $K$, for each label $\lambda \in \mathscr{L}$, both the affine constraint $\mathrm{V}_{(\lambda,x)} = \mathrm{P}_\lambda$ and the affine constraint $\mathrm{B}_{(\lambda,x)} = \delta_l^\lambda$ where $\delta_l^{l'} = 1$ if $l = l'$, and $\delta_l^{l'} = 0$ otherwise.

- the abstract garbage collection is defined by using the projection operator that is defined in [47] to keep the constraints about the variable $\mathscr{X}_{V \cap X}$.

**Theorem 8.3.13.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

**8.3.2.2.2 Molecule abstraction** In the same way, for any finite $n$-tuple $(V_i)_{1 \le i \le n}$ of interfaces, we denote by $\mathscr{X}_{(V_i)}$ the set of variables $\{\mathrm{P}_\lambda^i \mid \lambda \in \mathscr{L},\ 1 \le i \le n\} \cup \{\mathrm{V}_{(\lambda,v)}^i \mid \lambda \in \mathscr{L},\ 1 \le i \le n,\ v \in V_i\} \cup \{\mathrm{B}_{(l,v)}^i\ l \in \mathscr{L},\ 1 \le i \le n,\ v \in V_i\}$. We also define $Molecule_{(V_i)}^{\mathrm{GLOB}}$ as the set of affine equality relations among the variables of $\mathscr{X}_{(V_i)}$. For all affine system $K$ in $Molecule_{(V_i)}^{\mathrm{GLOB}}$, we define the concretization $\gamma_{(V_i)}^{\mathrm{GLOB}}(K) \subseteq \Pi_{1 \le 1 \le n} Env_{V_i}^{\mathscr{M}}$ as the set of the marker/environment pair families $(id_i, E_i)_{1 \le i \le n}$ such that the assignment:

$$\left[\begin{array}{lcl} \mathrm{P}_\lambda^i & \to & |id_i|_\lambda, \\ \mathrm{V}_{(\lambda,v)}^i & \to & |id_v|_\lambda \text{ where } (l', id_v) = E_i(v)], \\ \mathrm{B}_{(l,v)}^i & \to & \delta_l^{l'} \text{ where } (l', id_v) = E_i(v)] \end{array}\right]$$

is a solution of $K$.

Abstract primitives are defined as follows:

[10] $\delta_l^{l'} = 1$ if $l = l'$, and $\delta_l^{l'} = 0$ otherwise.

- To compute the affine system $\textsc{inj}^{\text{GLOB}}(K)$, we just replace each variable of the form $\textsc{p}_{\lambda}$ into the variable $\textsc{p}^{1}_{\lambda}$, each variable of the form $\textsc{v}_{(\lambda,v)}$ with the variable $\textsc{v}^{1}_{(\lambda,v)}$ in the affine system $K$, and each variable of the form $\textsc{b}_{(l,v)}$ with the variable $\textsc{b}^{1}_{(l,v)}$ in the affine system $K$.

- the abstract concatenation is computed by first renaming each variable in the second system, and then gathering the constraints. More precisely, let $m$, $n$ be two integers. Let $(U_i)_{1\leq i\leq m}$, $(V_i)_{1\leq i\leq n}$ be two tuples of interfaces. Let $K_1 \in \mathit{Molecule}^{\text{GLOB}}_{(U_i)}, K_2 \in \mathit{Molecule}^{\text{GLOB}}_{(V_i)}$ be two affine equality systems. The system $K_1 \bullet^{\text{GLOB}} K_2$ is defined as the affine system that contains all the constraints in $K_1$ and all the constraints in the system $K'_2$, where the affine system $K'_2$ is given by replacing each occurrence of variable $\textsc{p}^{i}_{\lambda}$ with the variable $\textsc{p}^{i+m}_{\lambda}$, each occurrence of variable $\textsc{v}^{i}_{(\lambda,v)}$ with the variable $\textsc{v}^{i+m}_{(\lambda,v)}$, and each occurrence of variable $\textsc{b}^{i}_{(l,v)}$ with the variable $\textsc{b}^{i+m}_{(l,v)}$.

- the abstract projection just keeps information about a given thread and renames the corresponding variables. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $k$ be an integer such that $1 \leq k \leq n$. The system $\textsc{proj}^{\text{GLOB}}(k,K)$ is obtained by applying the affine projection to keep information only about the variables of the form $\textsc{p}^{k}_{\lambda}$ (for any label $\lambda \in \mathscr{L}$), of the form $\textsc{v}^{k}_{(\lambda,v)}$ (for any label $\lambda \in \mathscr{L}$, any variable $v \in V_k$), or of the form $\textsc{b}^{k}_{(l,v)}$ (for any label $l \in \mathscr{L}$, any variable $v \in V_k$), and then by replacing each variable $\textsc{p}^{k}_{\lambda}$ with the variable $\textsc{p}_{\lambda}$, each variable $\textsc{v}^{k}_{(\lambda,v)}$ with the variable $\textsc{v}_{(\lambda,v)}$, and each variable $\textsc{b}^{k}_{(l,v)}$ with the variable $\textsc{b}_{(l,v)}$.

- the abstract extension is the system that is obtained by keeping only information about the variables that are not redefined and then by inserting redefined variables without any constraint about them. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $X$ be a subset of $\mathscr{V} \times [\![1;n]\!]$. Let $K$ be an element of $\mathit{Molecule}^{\text{GLOB}}_{(V_i)}$. We define the system $\textsc{new}^{\text{GLOB}}_{\top}(X,K) \in \mathit{Molecule}^{\text{GLOB}}_{(V_i\cup\{v \mid (v,i)\in X\})}$ as the affine system that is obtained by collecting all the constraints that only involve variables in $\mathscr{X}_{(V_i)} \setminus X$. (We use Gaussian elimination). Variables $\textsc{v}^{i}_{(\lambda,v)}$ and $\textsc{b}^{i}_{(l,v)}$ when $(v,i) \in X$ are tied with no constraint in the affine system $\textsc{new}^{\text{GLOB}}_{\top}(X,K)$.

- the abstract synchronization of an affine equality constraint system $K$ with respect to a constraint set $S$ consists in inserting some equality constraints. Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program point labels. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces. Let $S$ be a set of constraints $(x,k) \diamond (y,l)$ with $1 \leq k \leq n$, $x \in V_k \cup \{I\}$, $1 \leq l \leq n$, and $y \in V_l \cup \{I\}$. Let $K$ be an affine system in $\mathit{Molecule}^{\mathrm{GLOB}}_{(V_i)}$. We associate each formal variable $X$ and each label $\lambda \in \mathscr{L}$ with the affine combination pair $\mathit{aff}(X,\lambda)$ where:

$$\mathit{aff}(X,\lambda) = \begin{cases} (\delta_{\lambda}^{p^k}, \mathrm{P}^k_{\lambda}) & \text{if } X \text{ matches } (I,k),\\ (\mathrm{B}^k_{(\lambda,v)}, \mathrm{V}^k_{(\lambda,v)}) & \text{if } X \text{ matches } (v,k) \text{ with } v \in V_k,\end{cases}$$

where $\delta_l^{l'} = 1$ if $l = l'$ and $\delta_l^{l'} = 0$ otherwise.

We define $\mathrm{SYNC}^{\mathrm{GLOB}}(S,(p_i),K)$ as the affine system $K$ where we have inserted the constraints $\mathit{aff}(X,\lambda) = \mathit{aff}(Y,\lambda)$ for any label $\lambda \in \mathscr{L}$ and any equality constraint $(X = Y) \in S$. Gaussian elimination allows for the normalization of the result.

- marker allocation $\mathrm{FETCH}^{\mathrm{GLOB}}((p^i),\mathscr{K})$ is obtained by first using affine projection to keep only constraints not involving variables of the form $\mathrm{P}^1_{\lambda}$, and then by inserting the constraints $\mathrm{P}^1_{\lambda} = \mathrm{P}^2_{\lambda} + \delta^{\lambda}_{p^2}$ for any $\lambda$ in $\mathscr{L}$ (where $\delta^x_y$ is either equal to 1 if x=y, or equal to 0 otherwise).

**Theorem 8.3.14.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

#### 8.3.2.2.3 Invariant example

**Example 8.3.15.** *We analyze the mobile system that is given in Fig. 8.5 on page 223. The global numerical abstraction detects that:*

- *in each thread labeled* **2**, *the variable* ***right*** *is linked to a name created by the* $(\nu$ ***right***$)$ *restriction, while the variable* ***left*** *is linked to a name, either created by an instance of the* $(\nu$ ***right***$)$ *restriction, or by an instance of the* $(\nu$ **left0**$)$ *restriction. We also detect that, in the case where the variable* ***left*** *is linked to a name created by the* $(\nu$ ***right***$)$ *restriction, this variable is linked to the name created by the previous recursive instance of the one which has created the name communicated to the variable* ***right****;*

- *in each thread labeled* **5**, *the variable* **left0** *is linked to a name created by the* $(\nu$ **left0**$)$ *restriction, while the variable* ***left*** *is linked to a name either created by an instance of the* $(\nu$ ***right***$)$ *restriction or by an instance of the* $(\nu$ **left0**$)$ *restriction.*

*These properties are deduced from the following invariants:*

$$\begin{cases} f(2) \textit{ satisfies} \begin{cases} \mathrm{B}_{(\textit{right},\textit{right})} = 1 \\ \mathrm{B}_{(\mathrm{left0},\textit{left})} + \mathrm{B}_{(\textit{right},\textit{left})} = 1 \\ \mathrm{V}_{(3,\textit{right})} = \mathrm{V}_{(3,\textit{left})} + \mathrm{B}_{(\mathrm{right},\textit{left})} \end{cases} \\ f(5) \textit{ satisfies} \begin{cases} \mathrm{B}_{(\textit{right},\textit{left})} + \mathrm{B}_{(\mathrm{left0},\textit{left})} = 1 \\ \mathrm{B}_{(\mathrm{left0},\mathrm{left0})} = 1 \end{cases} \end{cases}$$

*where $f$ denotes the result of the analysis.*

*Nevertheless, our abstract domain is not expressive enough to merge these two environments, and detects no insightful information for the thread labeled* **9**. *That is why we introduce a partitioned domain.* □

#### 8.3.2.3 Partitioned numerical abstraction

We propose to partition the set of interactions between values and threads in order to get more accurate results. To avoid complexity explosion, we do not globally abstract environments, we only compare pair-wisely the right comb of the markers. Let $\psi$[11] be a linear form defined on $\mathbb{Q}^{\Sigma}$. The choice of $\psi$ depends on the cycles that are encountered in the graphs that are computed during the shape analysis. Counting the occurrence of each letter is almost the most costly choice, but it always lead to the best accuracy. When graphs only contains a single cycle, considering the length words is enough to get good results.

**8.3.2.3.1 Atom abstraction** Let $V \subseteq \mathscr{V}$ be a finite set of variables. We introduce the abstract domain $Atome_V^{\mathrm{PART}}$ of the functions which map the set $(V \times \mathscr{L}) \uplus ((V \times \mathscr{L})^2)$ onto the set of affine subspaces of $\mathbb{Q}^2$. For all $f \in Atome_V^{\mathrm{PART}}$, the concretization $\gamma_V^{\mathrm{PART}}(f) \in \wp(Env_V^{\mathscr{M}})$ is the set of pairs marker/environment $(id, E)$ such that:

- $\forall x \in V$, such that $E(x) = (c_x, id_x)$,
  $(\psi([\lambda \to |\phi_2(id)|_\lambda]), \psi([\lambda \to |\phi_2(id_x)|_\lambda])) \in f(x, c_x)$;
- $\forall x \in V, \forall y \in V$, such that $E(x) = (c_x, id_x)$ and $E(y) = (c_y, id_y)$,
  $(\psi([\lambda \to |\phi_2(id_x)|_\lambda]), \psi([\lambda \to |\phi_2(id_y)|_\lambda])) \in f((x, c_x), (y, c_y))$.

Because of the cost of the partitioning, we cannot afford much calculi in this domain. This domain will be used to locally refine (by partitioning) the information

[11]This abstraction must be done with several linear forms chosen according to a pre-analysis and Thm. 8.3.22.

we got from the global numerical abstraction GLOB before testing both matching guards and synchronization constraints.

The structure $(\sqsubseteq_V^{\text{PART}}, \sqcup_V^{\text{PART}}, \bot_V^{\text{PART}})$ is defined applying affine operators described in [47] component-wise. Since there is no infinite increasing sequences in $Atome_V^{\text{PART}}$, we define $a \nabla_V^{\text{PART}} b = \sqcup_V^{\text{PART}}\{a,b\}$. Other primitives are defined as follows:

- the empty environment abstraction is defined as follows:

$$\varepsilon^{\text{PART}} = \emptyset.$$

- when computing an abstract restriction, we take into account the fact that the marker of the new value is also the marker of the thread that creates this value. We ignore any relation between the other variables. These relations will be computed by reduction (Cf. Sect. 8.3.2.3.3). The abstract restriction is defined as follows:

$$\nu^{\text{PART}}(x,l,f) = f\left[\begin{array}{ccll} (x,l) & \mapsto & \{(n,n)\in\mathbb{Q}\}, & \\ (x,l') & \mapsto & \emptyset, & \text{if } l \neq l', \\ (x,l),(x,l) & \mapsto & \{(n,n)\in\mathbb{Q}\}, & \\ (x,l),(y,\_) & \mapsto & \mathbb{Q}^2, & \text{if } x \neq y, \\ (y,\_),(x,l) & \mapsto & \mathbb{Q}^2, & \text{if } x \neq y, \\ (x,l'),(\_,\_) & \mapsto & \emptyset, & \text{if } l \neq l', \\ (\_,\_),(x,l') & \mapsto & \emptyset, & \text{if } l \neq l' \end{array}\right]$$

- the abstract garbage collection consists in throwing away all the constraints that tie some variables that are garbage collected. Thus we define:

$$\text{GC}^{\text{PART}}(X,f) = f_{|(X\times\mathscr{L})\uplus(X\times\mathscr{L})^2}.$$

**Theorem 8.3.16.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

#### 8.3.2.3.2 Molecule abstraction

For any finite $n$-tuple $(V_i)_{1\leq i\leq n}$ of interfaces, we define the abstract domain $Molecule_{(V_i)}^{\text{PART}}$ as the Cartesian product $\Pi(Atome_{V_i}^{\text{PART}})$. For all tuple $(t_i)$ in $Molecule_{(V_i)}^{\text{PART}}$, we define the concretization $\gamma_{(V_i)}^{\text{PART}}((t_i)) \subseteq \Pi_{1\leq 1\leq n} Env_{V_i}^{\mathscr{M}}$ as the set of marker/environment pair families $\Pi\gamma_{V_i}^{\text{PART}}(t_i)$.

Abstract primitives are defined as follows:

- the abstract injection of an abstract atom $f \in Atome_V^{\text{PART}}$ is the 1-tuple $(f) \in Molecule_{(V)}^{\text{PART}}$.

- the abstract concatenation is the usual tuple concatenation. Let $m$, $n$ be two integers. Let $(U_i)_{1\leq i\leq m}$, $(V_i)_{1\leq i\leq n}$ be two tuples of interfaces. Let $f\in Molecule^{\text{PART}}_{(U_i)}, g\in Molecule^{\text{PART}}_{(V_i)}$ be two abstract molecules. The abstract molecule $f\bullet^{\text{PART}} g$ is defined as follows:
$$(f\bullet g)_i=\begin{cases} f_i & \text{if } 1\leq i\leq m\\ g_{i-m} & \text{if } m+1\leq i\leq m+n.\end{cases}$$
- the abstract projection is the usual component extraction. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $k$ be an integer such that $1\leq k\leq n$. The system $\text{PROJ}^{\text{PART}}(k,(f_i))$ is given by $f_k$.
- the abstract extension is the system that is obtained by keeping only information about the variables that are not redefined and then by inserting redefined variables without any constraint about them. Let $n$ be an integer and $(V_i)_{1\leq i\leq n}$ be a family of interfaces. Let $X$ be a subset of $\mathscr{V}\times[\![1;n]\!]$. Let $(f_i)$ be an element of $Molecule^{\text{PART}}_{(V_i)}$. We define the tuple $\text{NEW}^{\text{PART}}_{\top}(X,(f_i))\in Molecule^{\text{PART}}_{(V_i\cup\{v\mid(v,i)\in X\})}$ as the abstract element:
$$\left(f_i\left[\begin{array}{ccc|c} (v,\_) & \mapsto & \mathbb{Q}^2, & \\ ((v,\_),(\_)) & \mapsto & \mathbb{Q}^2, & v\in V_i,\text{ such that }(v,i)\in X\\ ((\_),(v,\_)) & \mapsto & \mathbb{Q}^2 & \end{array}\right]\right)_{1\leq i\leq n}$$
- we now define the abstract synchronization. We want to pass any relation among the values of two variables to any pair of variables that are simultaneously synchronized with these two variables. Let $n$ be an integer. Let $(p_i)$ be an $n$-tuple of program point labels. Let $(V_i)_{1\leq i\leq n}$ be an $n$-tuple of interfaces. Let $S$ be a set of constraints $(x,k)\diamond(y,l)$ with $1\leq k\leq n$, $x\in V_k\cup\{I\}$, $1\leq l\leq n$, and $y\in V_l\cup\{I\}$. For any variable $X\in\{(x,k)\mid 1\leq k\leq n,\ x\in V_k\cup\{I\}\}$, we denote by $[X]$ the equivalence class of $X$ in the set $\{(x,k)\mid 1\leq k\leq n,\ x\in V_k\cup\{I\}\}$ with respect to the reflexive, symmetric, and transitive closure of the relation $\mathscr{R}$ that is defined by $X\mathscr{R}Y$ if and only if $(X=Y)\in S$. Moreover, we say that two formal variables $(x,k)$ and $(y,l)$ constrain the same thread if and only if $k=l$. In such a case, we write $(x,k)\equiv(y,l)$. We define $\text{SYNC}^{\text{PART}}(S,(p_i),(f_i))$ as the tuple $(g_i)$ where for any $i$ such that $1\leq i\leq n$, the affine space $g_i$ is defined as follows:
$$\begin{cases}(v,l)\mapsto f_i(v,l)\cap\left(\bigcap\{h((X,p_i),(Y,l))\mid X\mathscr{R}(I,i),\ Y\mathscr{R}(v,i),\ X\equiv Y\}\right)\\ (v,l),(v',l')\mapsto f_i((v,l),(v',l'))\cap\left(\bigcap\left\{h((X,l),(Y,l'))\ \middle|\ \begin{array}{l}X\mathscr{R}(v,i),\\ Y\mathscr{R}(v',i),\\ X\equiv Y\end{array}\right\}\right)\end{cases}$$

where the function $h$ associates each pair of formal variable/label pair $((X_1,l_1),(X_2,l_2))$ with the abstraction of the marker pair $(id_1,id_2)$ such that $(l_1,id_1)$ may be the value of $X_1$ while $(l_2,id_2)$ is the value of $X_2$.The function $h$ is defined as follows:

$$h : \left\{ \begin{array}{rcll} (((I,i),l),\_) & \mapsto & \emptyset & \text{if } l \neq p_i, \\ (\_,((I,i),l)) & \mapsto & \emptyset & \text{if } l \neq p_i, \\ ((X,l),(X,l)) & \mapsto & \{(n,n) \mid n \in \mathbb{Q}\}, & \\ (((I,i),p_i),((v,i),l)) & \mapsto & f_i(v,l), & \\ (((v,i),l),((I,i),p_i)) & \mapsto & \{(n,m) \mid (m,n) \in (f_i(x,l))\}, & \\ (((v,i),l),((v',i),l')) & \mapsto & f_i((v,l),(v',l')), & \\ \_ & \mapsto & \mathbb{Q}^2. & \end{array} \right.$$

Special care is to be taken in order to deal with the formal variables of the form $(I,i)$ (which denotes a thread identity).

- marker allocation $\text{FETCH}^{\text{PART}}((p^i),(f_i))$ is given by replacing the first component by the function that maps any argument to $\mathbb{Q}^2$.

**8.3.2.3.3 Reduction** We shall notice that no information is computed by abstract restriction and abstract marker allocation. Nevertheless, during synchronization, the description of the communicated names is copied to the description of the receiver environment. We use a reduction operator to restore missing constraints. A complete reduction of properties would lead to a time complexity explosion. We use a partial reduction. On the first hand, we use thread markers as pivots and replace each abstract atom $f$ with the following element:

$$f\left[((x,c),(y,d)) \rightarrow f((x,c),(y,d)) \cap \left\{ (x,y) \;\middle|\; \exists z, \left\{ \begin{array}{l} (z,y) \in f(y,d), \\ (z,x) \in f(x,c) \end{array} \right\} \right\} \right],$$

and on the other hand, we always perform reductions between global numerical abstraction and partitioned numerical abstraction: global numerical analysis is used to collect all the information, which is then projected onto each case of the partition (Cf. Sect. 8.3.3).

The reduction over abstract molecules is defined component-wise.

#### 8.3.2.3.4 Invariant example

**Example 8.3.17.** *For $\psi$, we choose the linear form which maps each vector to the sum of its components. Our analysis succeeds in proving that the second* *pattern matching, in the example, is not satisfiable. Along the abstract iteration, the analyzer proves that in thread* **9***, the names linked to the variables* **x** *and* **y** *have been respectively declared:*

1. *by the action* $(\nu\ \mathbf{left0})$ *of a thread with a* **0** *marker length and the action* $(\nu\ \mathbf{left0})$ *of a thread with a* **0** *marker length;*

2. *or by the action* $(\nu\ \mathbf{left0})$ *of a thread with a* **0** *marker length and by the action* $(\nu\ \boldsymbol{right})$ *with a* **1** *marker length;*

3. *or by the action* $(\nu\ \boldsymbol{right})$ *of a thread* $t_1$ *and by the action* $(\nu\ \boldsymbol{right})$ *of a thread* $t_2$ *such that the length of the marker of* $t_2$ *is equal to the length of the marker of* $t_1$ *plus* **1***;*

4. *or by the action* $(\nu\ \boldsymbol{right})$ *of a thread of arbitrary marker length and by the action* $(\nu\ \mathbf{left0})$ *of a thread the length of the marker of which was* **0***.*

*Then it detects that the matching pattern* $([\mathbf{x} = \mathbf{y}])$ *may only be satisfied for the case 1 and it discovers that in thread* **7***, all the syntactic channel names* **x***,* **y** *and* **left0** *are bound to a channel created by the action* $(\nu\ \mathbf{left0})$ *the thread marker of which is* $\varepsilon$ *and concludes that the second pattern matching* $([\mathbf{x} \neq \mathbf{left0}])$ *may not be satisfied.* □

### 8.3.3 Approximated reduced product

These five domains collaborate in order to refine abstract constraints. We build our main abstraction as the product (Cf. Sect. 7.3.1 on page 7.3.1) of the five abstraction $=_1$, $=_2$, SHAPE, GLOB, and PART. We use a reduction operator $\rho$ to refine the constraints of each others. As explained in Sect. 7.3.2 on page 161, this reduction operator will be applied both at the beginning of an abstract computation step (to refine preconditions) and at the end of an abstract computation step (to refine newly computed abstract values). Moreover, in order to get more accuracy, we will apply this reduction after any synchronization, in order to disable as many interactions as possible. All abstract primitives are defined component-wise, except the abstract synchronization that first consists in computing abstract synchronization component-wise and then in applying the reduction on the result.

#### 8.3.3.1 Reduction steps

We define the reduction by using a set of rules. Each rule describes how a domain reduces the information of another domain. A graphical summary of all these reductions is given in Fig. 8.6. We now display the set of reduction rules:

1. we use the information about equality of values and equality of markers to refine marker analysis; conversely, we use the information obtained in marker analysis to infer information about disequalities of values and disequalities of markers;

2. we make reductions between the global and partitioned numerical abstractions: for the sake of simplicity, we take the case of abstract atoms over the interface $V$. Let $K$ be a global approximation and $f$ be a partitioned approximation.

   - We want to refine the image $f((v,l),(v',l'))$: First we compute a new affine system $K'$ by inserting in $K$ for each $\lambda \in \mathscr{L}$, the two constraints $\mathrm{B}_{(\lambda,v)} = \delta_l^\lambda$ and $\mathrm{B}_{(\lambda,v')} = \delta_{l'}^\lambda$, and by collecting (*via* Gaussian elimination) all the equations that constrain only the variables $\mathrm{V}_{(x,\lambda)}$ for each $x \in \{v;v'\}$ and each $\lambda \in \mathscr{L}$. We can soundly replace $f((v,l),(v',l'))$ with $f((v,l),(v',l')) \cap \{(\psi(u),\psi(v)) \mid [\mathrm{V}_{(v,\lambda)} \mapsto |u|_\lambda,\ \mathrm{V}_{(v',\lambda)} \mapsto |v|_\lambda]$ satisfies $K\}$.
   - Conversely, we can refine the affine system $K$: Let $v$ and $v'$ be two variables in $V \cup \{I\}$. For the sake of simplicity, we suppose that $v \neq I$ and $v' \neq I$. We define the affine system $K'(v,v')$ as the affine hull $\sqcup_{\mathscr{X}_V}^{\mathscr{K}} \{J(v,l,v',l') \mid (l,l') \in \mathscr{L}^2\}$ where for each label $l,\ l' \in \mathscr{L}$, the affine system $J(v,l,v',l')$ is given by the constraints $\mathrm{B}_{(\lambda,v)} = \delta_l^\lambda$, $\mathrm{B}_{(\lambda,v')} = \delta_{l'}^\lambda$, and $(\psi([\lambda \mapsto \mathrm{V}_{(\lambda,v)}]), \psi([\lambda \mapsto \mathrm{V}_{(\lambda,v')}])) \in f((v,l),(v,l'))$ for any $\lambda \in \mathscr{L}$. We can soundly replace the system $K$ with the system that collects all the constraints in the system $K$ and all the constraints in any system $K'(v,v')$.

3. We use the results obtained in marker analysis to prove equality relations between name markers. This reduction step is the more complex. It is the goal of Sect. 8.3.3.2.

#### 8.3.3.2 Parikh's vector equality and word equality

We now give some details about the last kind of reductions. Marker analysis may discover that two markers are recognized by the same automaton $\mathscr{A}$ and have the same Parikh's vector, but these two conditions do not ensure that these two markers are the same. We give in Thm. 8.3.22 a decidable sufficient condition on the automaton $\mathscr{A}$ to ensure the equality of such markers.

Let $\Sigma$ be a finite alphabet, $\phi$ be a linear function from the vector space $\mathbb{R}^\Sigma$ into the vector space $\mathbb{R}^m$, and $\mathscr{A}$ be an automaton $(Q,\rightarrow,i,f)$ such that the set $Q$ is finite, the relation $\rightarrow$ is a part of $Q \times \Sigma \times Q$ and $i$ and $f$ are parts of $Q$.

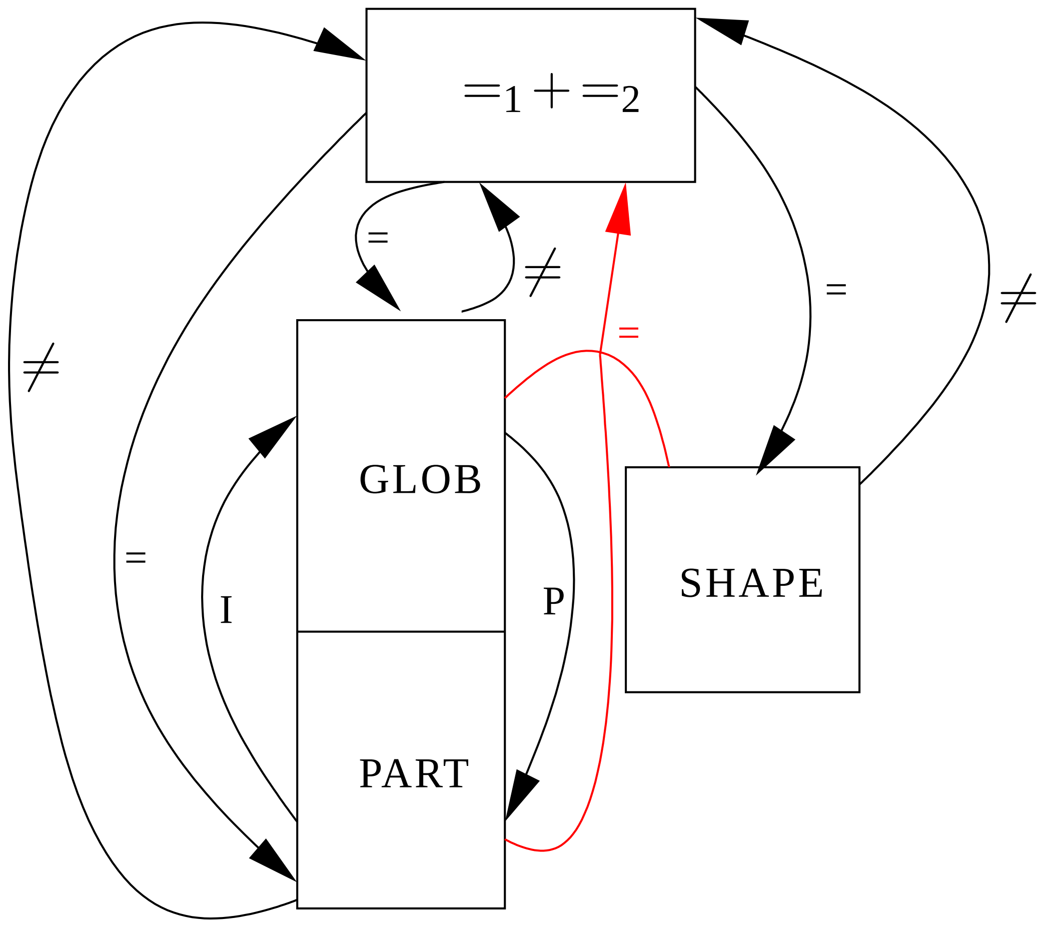


Figure 8.6: Reduction graph.

**Definition 8.3.18.** We define the set $Path(\mathscr{A})$ of acyclic derivation sequences in $\mathscr{A}$ as the set of sequences $q_0 \overset{\lambda_1}{\to} \dots \overset{\lambda_n}{\to} q_n$ such that $q_0 \in i$, $q_n \in f$, for all $i, j \in [\![0;n]\!]$, $i \neq j \implies q_i \neq q_j$, and for all $i \in [\![1;n]\!], (q_{i-1}, \lambda_i, q_i) \in \to$.

**Definition 8.3.19.** Let $q$ be a state in $Q$. We define the set $Cycle(\mathscr{A}, q)$ of elementary cycles of $\mathscr{A}$ stemming from the state $q$, as the set of sequences $q = q_0 \overset{\lambda_0}{\to} q_1 \dots q_n \overset{\lambda_n}{\to} q_{n+1} = q$ such that for all $i \in [\![1;n]\!]$, $q_i \neq q$, and for all $i \in [\![0;n]\!]$, $(q_i, \lambda_i, q_{i+1}) \in \to$.

**Definition 8.3.20.** Let $q_0 \overset{\lambda_1}{\to} \dots \overset{\lambda_n}{\to} q_n$ be a sequence in $\mathscr{A}$, we define its affine description $\mathscr{P}(q_0 \overset{\lambda_1}{\to} \dots \overset{\lambda_n}{\to} q_n)$ as the vector $\phi([\lambda \to |\lambda_1 \dots \lambda_n|_\lambda])$.

**Definition 8.3.21.** Let $a = q_0 \overset{\lambda_1}{\to} \dots \overset{\lambda_n}{\to} q_n$ be an acyclic derivation sequence in $Path(\mathscr{A})$, we define the family $\mathscr{F}(a)$ of affine descriptions of the cycles of the derivation $a$ as the family $(\mathscr{P}(c))_{c \in \bigcup\{Cycle(q_i) \mid i \in [\![0;n]\!]\}}$.

**Theorem 8.3.22.** *If*

1. *for all $q \in Q$, $Card(Cycle(\mathscr{A}, q)) \leqslant 1$,*
2. *for all $a \in Path(\mathscr{A})$, $\mathscr{F}(a)$ is linearly independent in $\mathscr{R}^m$,*
3. *for all distinct acyclic derivations $a, a' \in Path(\mathscr{A})$, the two affine sets $\mathscr{P}(a) + \mathrm{Vect}(\mathscr{F}(a))$ and $\mathscr{P}(a') + \mathrm{Vect}(\mathscr{F}(a'))$ are disjoint,*

*then:*

$$\forall u, v \in \Sigma^*,\ [u, v \textit{ recognized by } \mathscr{A} \textit{ and } \phi[\lambda \to |u|_\lambda] = \phi[\lambda \to |v|_\lambda]] \implies u = v.$$

Roughly speaking, the first condition ensures that the automaton $\mathscr{A}$ contains no embedded cycle. Then, from the description of a word Parikh's vector, we can deduce which main acyclic derivation is to be used to recognize a word (third condition), and how many times each cycles are to be used (second condition).

# Chapter 9

# Occurrence number approximation

We now propose to count the occurrence number of threads during computation sequences. It is especially useful to detect mutual exclusion. It also helps in discovering sound bounds for the number of threads during computation sequences, so that we can verify that some part of the system will not exceed the physical limits imposed by the implementation of the system. In the case of the *ftp*-server, we can automatically infer the maximum number of simultaneous client sessions.

We first abstract away any information about markers and environments. This way, a configuration is just seen as a multi-set of syntactic threads. Our approach relies on the use of a reduced product between a non-relational and a relational domain. Complexity issues are solved by using approximate algorithms for computing a reduction between these two domains. Moreover, there may be no relation among the occurrence number of some syntactic threads, whereas the occurrence number of these threads may be related to the number of the performed transitions. For instance, the occurrence number of a given syntactic thread is usually related with the occurrence number of the syntactic threads in its potential continuations. But in the case of an asynchronous computation, all continuations for the syntactic thread are empty. Nevertheless, we can infer a relationship between the occurrence number of the thread and the number of computations of a given transition; then a bound for this transition number gives a bound for the occurrence number of the computed thread (cf Remark 9.4.3 on page 260).

Our abstraction requires an abstract domain to describe families of natural numbers. In Sect. 9.2, we introduce a generic analysis, independently of this domain. In Sect. 9.3, we propose a well-suited abstract domain. In Sect. 9.4, we give experimental results. This chapter extends the framework [38] to our meta language.

## 9.1 Related works

Several existing analyses use properties about the occurrence number of threads in order to refine control flow properties. In [51, 50], Levi and Maffeis propose a uniform control flow analysis for mobile *ambients*, where a counter is associated to each program point. This counter describes whether several instances of threads at the same program point may occur simultaneously, or not. At the concrete level, an ambient may not migrate into itself. If the counter associated to the program point $p$ indicates that there may be at most one simultaneous instance of thread at program point $p$, then we can discard at the abstract level the migration of ambients created at program $p$ into ambients created at program point $p$. Nevertheless, this analysis is not relational and it captures no property about the program points that are bellow a bang operator in the initial state. This way, it gives no information about systems of the form $!P$.

In [60], Nielson *et al.* refine a control flow analysis for BIO-*ambients*, by detecting some mutual exclusion. They take care about recursion and internal choices. Their mutual exclusion analysis detects the pairs $(p,q)$ of program points such that no instance of thread at program point $p$ may interact with an instance of tread at program point $q$ because such two instances are separated by an internal choice in any recursion unfolding of the syntactic description of the initial state. Here again it is only a syntactic criterion (up to recursion unfolding). It gives no information about systems of the form $\mathrm{let}^p\ A^\alpha\langle\rangle = P \mid A^q\langle\rangle \text{ in } P$. This analysis is not control-flow sensitive, whereas ours considers only the interactions that are detected by the control flow analysis.

Finite-control system characterizations have been made, because model checking finite-control systems are quite easy. A finite-control $\pi$-calculus [27] is obtained by disallowing parallel composition through bang or recursion. Charatonik *et. al* have proposed a type system [19] for the mobile ambients with recursion to ensure that mobile systems have a finite control. This type system counts the number of threads that must be consumed to unfold recursions, in order to prove that the system never grows up between two unfolding of the same recursion. Finite-control systems are very restrictive. Our analysis detects some parts of the system that may launch only a bounded number of simultaneous threads and captures an approximation of the part of the system that may launch an unbounded number of simultaneous threads, whereas finite-control type systems reject any system that contains an unbounded part. Since finite-control characterizations are defined by induction over the syntax, they are control-flow insensitive, whereas our analysis interacts with our environment analysis.

Very few relational analyses for counting occurrences of threads have been published. Nevertheless, this problem is very close to the problem of approximating the behavior of a Petri net, and of occurrence counting in mobile *ambients*.

In [43], Nielson *et al.* propose an exponential analysis for counting occurrences of threads inside ambients. In [63], they use context-dependent counts for inferring a more accurate description of the internal structure of threads, at the expense of a higher time complexity (an exponential number of threads are distinguished). These analyses rely on the use of a non-relational domain to abstract the content of an ambient. Then, they use disjunctive completion, and abstract any potential content of a syntactic ambient in the power set of this abstract domain. These two analyses encounter the same problem: in case several instances of the same thread may coexist, when one instance of this thread performs a computation step, these analyses may not decide whether only one or several instances remain after this computation step, so they have to consider both cases, which leads to both a loss of precision and an exponential explosion in complexity. The use of an approximated reduced product between a relational domain and a non-relational domain to globally abstract sets of multi-sets of threads allows us to solve this problem efficiently. Thus, we obtain a very accurate analysis which is polynomial in the number of program points (i.e. polynomial in the size of the initial system configuration).

## 9.2 Generic analysis

We recall that the set $\mathscr{L}_{\mathrm{p}}$ is the set of the labels of the program points of system $\mathscr{S}$. We choose a set $\Sigma_{sub}$ of variables in order to count the transitions that are performed. Since the number of transition labels may be huge (for instance, it is quadratic with respect to the number of program points in the case of the $\pi$-calculus), we may want to quotient this set into equivalence classes, in order to deal with fewer variables. For that purpose, we relate the set $\Sigma$ to the smaller set $\Sigma_{sub}$ by an onto map $\psi$.

**Example 9.2.1.** *In the case of the $\pi$-calculus, we propose three natural choices for $\Sigma_{sub}$ and $\psi$ according to the expected trade-off between accuracy and efficiency: let $\phi$ be the function that associates each label of transition $(\mathscr{R},((t_?,pi_?),(t_!,pi_!)))$ onto the pair $(t_?,t_!)$ that gives the labels of the program points of the computing threads. We describe as follows the choice for $\Sigma_{sub}$ and $\psi$:*

1. *we can use one extra variable by setting $\Sigma_{sub} = \{0\}$ and $\psi(\lambda) = 0$,*

2. *we can use a linear number of extra variables by assigning $\Sigma_{sub} = \mathscr{L}_p$ and $\psi(\lambda) = snd(\phi(\lambda))$,*

3. *we can use a quadratic number of extra variables by assigning $\Sigma_{sub} = (\mathscr{L}_p)^2$ and $\forall i,j \in \mathscr{L}_p,\ \psi(\lambda) = \phi(\lambda)$.*

Let $\mathscr{V}_c$ be the set $\mathscr{L}_{\mathrm{p}} \cup \Sigma_{sub}$. We introduce an abstraction $\Pi_{\mathbb{N}^{\mathscr{V}_c}}$ which maps each concrete configuration $(u,C)$ in $\Sigma^* \times \mathscr{C}$ (we recall that $\mathscr{C}$ is the set of all the non standard configurations that satisfy marker unambiguity) to a family of natural numbers indexed by the set $\mathscr{V}_c$, as follows:

$$(\Pi_{\mathbb{N}^{\mathscr{V}_c}}(u,C))_v = \begin{cases} Card(\{(p,id,E) \in C \mid p = v\}) & \text{if } v \in \mathscr{L}_{\mathrm{p}} \\ \sum\{|u|_\lambda \mid \lambda \in \psi^{-1}(v)\} & \text{if } v \in \Sigma_{sub}. \end{cases}$$

**Proposition 9.2.2.** *Let $(u_1,C_1)$ and $(u_2,C_2)$ be two configurations such that $C_1 \cap C_2 = \emptyset$, then for any $v \in \mathscr{V}_c$, we have $\Pi_{\mathbb{N}^{\mathscr{V}_c}}(u_1.u_2, C_1 \cup C_2)_v = (\Pi_{\mathbb{N}^{\mathscr{V}_c}}(u_1,C_1))_v + (\Pi_{\mathbb{N}^{\mathscr{V}_c}}(u_2,C_2))_v$.*

Then, we consider $\wp(\mathbb{N}^{\mathscr{V}_c})$, the complete lattice of sets of natural number families indexed by $\mathscr{V}_c$. The power set $\wp(\mathbb{N}^{\mathscr{V}_c})$ is related to our concrete domain $\wp(\Sigma^* \times \mathscr{C})$ via a concretization function $\gamma_{\mathbb{N}^{\mathscr{V}_c}}$, where for any $A$ in $\wp(\mathbb{N}^{\mathscr{V}_c})$, the concretization $\gamma_{\mathbb{N}^{\mathscr{V}_c}}(A^\sharp)$ is defined as follows:

$$\left\{(u,C) \in \Sigma^* \times \mathscr{C} \;\middle|\; \Pi_{\mathbb{N}^{\mathscr{V}_c}}(u,C) \in A^\sharp \right\}.$$

We then introduce a generic pre-order $(\mathscr{N}_{\mathscr{V}_c}, \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}})$ to represent sets of natural number families indexed by $\mathscr{V}_c$. It is related to $\wp(\mathbb{N}^{\mathscr{V}_c})$ via a monotonic concretization $\gamma_{\mathscr{N}_{\mathscr{V}_c}}$. Furthermore, we introduce several generic primitives: a representation $\bot_{\mathscr{N}_{\mathscr{V}_c}}$ of the empty set, an abstract union $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}$, a widening operator $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$, an abstract counterpart $+^\sharp$ to the binary addition, an abstract counterpart $-^\sharp$ to the subtraction, an abstract synchronization $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}$. We also require the abstraction of some elementary families: an abstract element $0_{\mathscr{N}_{\mathscr{V}_c}}$ to represent the singleton containing the 0 family which associates 0 to each element in $\mathscr{V}_c$, and $\forall v \in \mathscr{V}_c$, an abstract element $1_{\mathscr{N}_{\mathscr{V}_c}}(v)$ to represent the singleton containing the family $\delta^v$ which associates 1 to the element $v$ and 0 to any other elements.

These primitives should satisfy the following properties:

1. $\gamma_{\mathscr{N}_{\mathscr{V}_c}}(\bot_{\mathscr{N}_{\mathscr{V}_c}}) = \emptyset$,

2. $\forall a \in \mathscr{N}_{\mathscr{V}_c}$, $\bot_{\mathscr{N}_{\mathscr{V}_c}} \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}} a$,

3. $\forall A \in \wp_{\text{finite}}(\mathscr{N}_{\mathscr{V}_c})$, $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}(A) \in \mathscr{N}_{\mathscr{V}_c}$ and $\forall a \in A$, $a \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}} \sqcup_{\mathscr{N}_{\mathscr{V}_c}}(A)$,

4. $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$ satisfies Def. 7.2.1.(7) on page 158,

5. $\forall a,b \in \mathscr{N}_{\mathscr{V}_c}$, $a +^\sharp b \in \mathscr{N}_{\mathscr{V}_c}$ and
   $\{(t_v + u_v)_{v \in \mathscr{V}_c} \mid t \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a),\, u \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(b)\} \subseteq \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a +^\sharp b)$,

6. $\forall a,b \in \mathcal{N}_{\mathcal{V}_c}$, $(a -^{\sharp} b) \in \mathcal{N}_{\mathcal{V}_c}$ and
$$\left\{ (t_v - u_v)_{v\in\mathcal{V}} \;\middle|\; \begin{array}{l} t \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a),\, u \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(b), \\ \forall v \in \mathcal{V}_c,\, t_v \geqslant u_v \end{array} \right\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\sharp} -^{\sharp} b^{\sharp}),$$

7. $\forall a \in \mathcal{N}_{\mathcal{V}_c}$, $t \in \mathbb{N}^{\mathcal{V}_c}$, $\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}(t, a^{\sharp}) \in \mathcal{N}_{\mathcal{V}_c}$ and
$\{u \mid u \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a), u_v \geq t_v,\, \forall v \in \mathcal{V}_c\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}(t,a))$,

8. $0_{\mathcal{N}_{\mathcal{V}_c}} \in \mathcal{N}_{\mathcal{V}_c}$ and $(0)_{i\in\mathcal{V}_c} \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(0_{\mathcal{N}_{\mathcal{V}_c}})$,

9. $\forall v \in \mathcal{V}_c$, $1_{\mathcal{N}_{\mathcal{V}_c}}(v) \in \mathcal{N}_{\mathcal{V}_c}$ and $(\delta_i^v)_{i\in\mathcal{V}_c} \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(1_{\mathcal{N}_{\mathcal{V}_c}}(v))$,
where $\delta_i^v = \begin{cases} 1 & \text{if } i = v \\ 0 & \text{otherwise.} \end{cases}$

Roughly speaking, the operator $+^{\sharp}$ is an abstract counterpart to the component-wise addition, the operator $-^{\sharp}$ is an abstract counterpart to the component-wise subtraction, and the primitive $\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}$ is used to extract from an abstract value the representation of the configurations which simultaneously contain all the threads required by a computation step.

We use these generic primitives to define some other primitives:

- We introduce the primitive $\sum^{\sharp}$ that computes the abstract sum of the element of a finite sequence of abstract elements. It is inductively defined by $\sum^{\sharp}(()_{i\in\emptyset}) = 0_{\mathcal{N}_{\mathcal{V}_c}}$ and by $\sum^{\sharp}((a_i)_{i\in I}) = a_{min(I)} +^{\sharp} \sum^{\sharp}\left((a_i)_{i\in I\setminus\{min(I)\}}\right)$ in the case where $I$ is a non-empty totally ordered set.

- We introduce the primitive $\chi^{\sharp}$ as the sound counterpart of the characteristic function. It maps each subset $V$ of $\mathcal{V}_c$ to the abstraction of the tuple which associates 1 to the variables in $V$ and 0 to the variables in $\mathcal{V}_c \setminus V$. The primitive $\chi$ is defined by $\chi^{\sharp}(V) = \sum^{\sharp}(1_{\mathcal{N}_{\mathcal{V}_c}}(v))_{v\in V}$.

We now define the abstract transition system. The initial configuration is defined in Fig. 9.1 and the abstract transition rule is given in Fig. 9.2. These definitions use an auxiliary primitive $\beta_{\mathcal{N}_{\mathcal{V}_c}}$ which computes an approximation of all potential continuations of a syntactic thread. This primitive allows the modeling of internal choices and parallel composition. The primitive $\beta_{\mathcal{N}_{\mathcal{V}_c}}$ is defined over the set of the syntactic continuation sets (cf. Def. 4.2.2 on page 54) as follows:

$$\beta_{\mathcal{N}_{\mathcal{V}_c}} : \begin{cases} \wp(\wp(\mathcal{L}_{\mathrm{p}} \times (\mathcal{V} \rightharpoonup \mathcal{L}))) & \to \mathcal{N}_{\mathcal{V}_c} \\ Ct & \mapsto \sqcup_{\mathcal{N}_{\mathcal{V}_c}}\{\chi^{\sharp}(fst(A)) \mid A \in Ct\}. \end{cases}$$

This way, each choice of continuation is abstracted by using the primitive $\chi^{\sharp}$, which models the parallel composition. All potential continuation abstractions are gathered by using the primitive $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}$: this model internal choices. The following proposition relates the concrete primitive for launching continuations *launch* (cf Def. 4.4.7 on page 59) and the abstract primitive $\beta_{\mathscr{N}_{\mathscr{V}_c}}$:

**Proposition 9.2.3 (abstract continuation computation).** *Let $Ct \in \wp(\wp(\mathscr{L}_p \times (\mathscr{V} \rightharpoonup \mathscr{L})))$ be a set of continuation choices.*
*Let $A \in Ct$ be a continuation. Let $id \in \mathscr{M}$ be a thread marker and let $E \in \mathscr{V} \rightharpoonup \mathscr{L} \times \mathscr{M}$ be an environment. We have:*

$$(\varepsilon, launch(A, id, E)) \in (\gamma_{\mathbb{N}^{\mathscr{V}_c}} \circ \gamma_{\mathscr{N}_{\mathscr{V}_c}})(\beta_{\mathscr{N}_{\mathscr{V}_c}}(Ct)).$$

The initial abstract configuration is obtained by applying $\beta_{\mathscr{N}_{\mathscr{V}_c}}$ to the initial system description $init_s$ (cf Sect. 4.2.4 on page 54). An intuitive explanation of abstract transitions is given in Fig. 9.3. We start from an abstract property $C^\sharp$ (on the left in the figure). We first compute an abstraction of the set of the configurations that satisfy the abstract property $C^\sharp$ and in which all interacting threads occur simultaneously (in dotted line in the figure). If this abstraction is the bottom element, the abstract interaction is disabled. Otherwise, the result $\overline{C}^\sharp$ of the interaction is obtained by translation: we decrease the occurrence number of the interacting thread which are not resources, and we increase the occurrence number related to both the description of continuations and the performed interaction. We do not bother about the control flow, since we use a Cartesian product between this analysis and one of our control flow analyses (cf Chap. 8).

**Theorem 9.2.4.** *$(\mathscr{N}_{\mathscr{V}_c}, \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}, \sqcup_{\mathscr{N}_{\mathscr{V}_c}}, \bot_{\mathscr{N}_{\mathscr{V}_c}}, \gamma_{\mathbb{N}^{\mathscr{V}_c}} \circ \gamma_{\mathscr{N}_{\mathscr{V}_c}}, C_0^{\mathscr{N}_{\mathscr{V}_c}}, \rightsquigarrow_{\mathscr{N}_{\mathscr{V}_c}}, \nabla_{\mathscr{N}_{\mathscr{V}_c}})$ is an abstraction.*

The proof of Thm. 9.2.4 is shown in appendix D.2.

## 9.3 Abstract domains

We only need to define an abstract domain to approximate sets of indexed families of natural numbers in which abstract primitives may be precisely and efficiently implemented. On the one hand, the primitive $\text{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}$ needs to express the property that several variables may simultaneously be greater than one, which is a relational information. On the other hand, the shape of the abstract transition rule suggests that the domain should be closed under addition. This problem is very similar to approximating the collecting semantics of a Petri net. We reject the use of usual numerical domains. We are unlikely to design a precise primitive $\text{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}$ in non-relational domains. Moreover, useful constraints are very likely to involve more than two variables, so that Miné's domains [58, 55, 56] may not

$$C_0^{\mathcal{N}_{\mathcal{V}_c}} = \beta_{\mathcal{N}_{\mathcal{V}_c}}(init_s)$$

Figure 9.1: Initial configuration.

Let $v^\sharp$ be an abstract configuration.
Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule.
Let $(p^k)_{1\leq k\leq n} \in C^n$ be an $n$-tuple of program point labels.
Let $(pi^k)_{1\leq k\leq n} = (s^k, (parameter_l^k)_l, (bd_l^k)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$ be a sequence of partial interactions.
We denote by $t \in \mathbb{N}^{\mathcal{V}_c}$ the tuple such that $t_v$ is the number of occurrence of $v$ in the sequence $(p_k)_{1\leq k\leq n}$.
If:

- $pi^k \in interaction(p_k)$, for any $k \in [\![1;n]\!]$,
- $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, v^\sharp) \neq \bot_{\mathcal{N}_{\mathcal{V}_c}}$,

then:

$$v^\sharp \overset{(\mathscr{R},(p^k,pi^k))}{\rightsquigarrow}_{\mathcal{N}_{\mathcal{V}_c}} \text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, v^\sharp) +^\sharp Transition +^\sharp Launched -^\sharp Consumed,$$

where:

- $Transition = 1_{\mathcal{N}_{\mathcal{V}_c}}(\psi(\mathscr{R}, (p^k, pi^k)))$;
- $Launched = \sum^\sharp(\beta_{\mathcal{N}_{\mathcal{V}_c}}(continuation^k))_{1\leq k\leq n}$;
- $Consumed = \sum^\sharp(1_{\mathcal{N}_{\mathcal{V}_c}}(p_k))_{k\in\{k' \mid 1\leq k'\leq n,\ type(s_{k'})\neq replication\}}$.

Figure 9.2: Abstract transition rule.

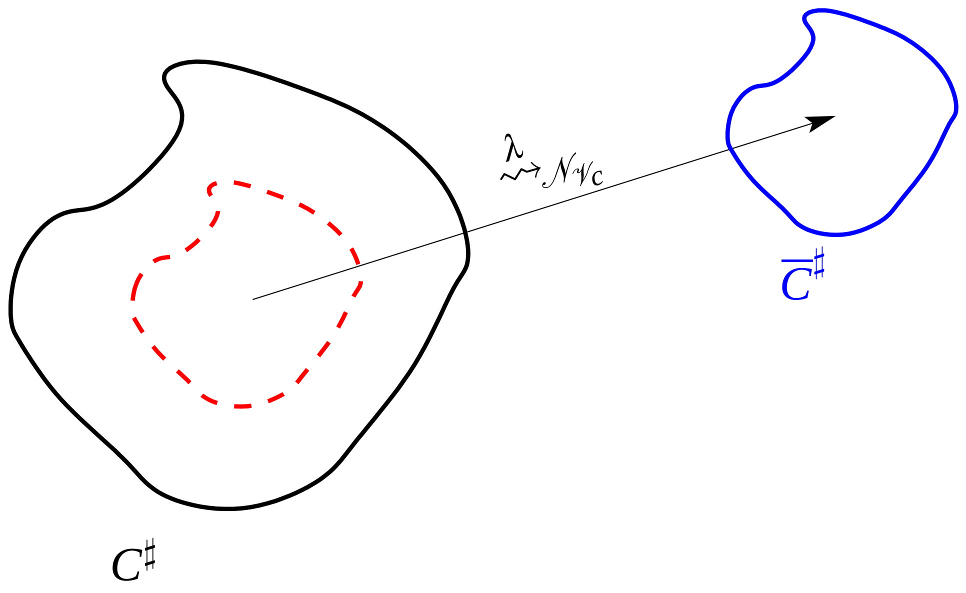

$\overset{\lambda}{\rightsquigarrow}_{\mathcal{N}\mathcal{V}_C}$
$\overline{C}^\sharp$
$C^\sharp$

Figure 9.3: Intuitive abstract transition.

help. Disjunctive completion may be used to lift a non-relational domain into a relational one: each configuration is abstracted in a finite non-relational domain, and then the abstraction of a configuration set is given by the collection of all the abstractions of its elements. Nevertheless, disjunctive completion often leads to a lack of accuracy and to an exponential explosion. Disjunctive completion has been used in [43,63] for abstracting the content of ambients. We cannot afford the domain of linear inequalities among a finite set of variables [26] because we deal with too many variables.

We propose the use of a reduced product between two domains: the interval lattice [22] and the linear equalities lattice [47].

- The first domain is used for expressing properties of interest. This domain may represent all the properties we need to express non-exhaustion of resources, but it may not calculate them precisely without being refined.

- The second domain is used for expressing more complex properties, such as mutual exclusion, which allows for more precise calculations in the first domain.

The power of our analysis directly comes from the use of an inexpensive algorithm, straightforwardly adapted from Linear Constraint Programming, to compute an approximated reduction between these two domains.

Recently, a new domain, the octahedron domain, has been proposed in [69]. This domain is expressive enough to deal with the constraints that interest us.

Nevertheless, this domain needs to be guided to avoid complexity explosion. So that, it cannot be applied straight forwardly, but it would be interesting to check how useful this domain may be for our application.

### 9.3.1 Interval domain

The complete lattice $(\mathscr{I}, \sqsubseteq_{\mathscr{I}}, \sqcup_{\mathscr{I}}, \bot_{\mathscr{I}}, \sqcap_{\mathscr{I}}, \top_{\mathscr{I}})$ is the functional domain of natural number intervals, where lattice operations are defined point-wisely. The abstract domain $\mathscr{I}$ is related to $\wp(\mathbb{N}^{\mathscr{V}_c})$ via the monotonic map $\gamma_{\mathscr{I}}$ defined by:

$$\gamma_{\mathscr{I}}(f) = \{u \in \mathbb{N}^{\mathscr{V}_c} \mid \forall i \in \mathscr{V}_c,\ u_i \in f(i)\}.$$

A family $(\nabla^n_{\mathscr{I}})$ of widening operators on $\mathscr{I}$ is defined as follows:

$$[f \nabla^n_{\mathscr{I}} g](x) = f(x) \nabla^n g(x),$$

where $[\![a;b]\!] \nabla^n [\![c;d]\!] = \begin{cases} [\![min\{a;c\};+\infty[\![ & \text{if } d > max\{b;n\} \\ [\![min\{a;c\};max\{b;d\}]\!] & \text{otherwise.} \end{cases}$

We can easily define abstract primitives in $\mathscr{I}$ as follows:

- $(f +_{\mathscr{I}} g) = [x \to f(x) + g(x)]$,
- $(f -_{\mathscr{I}} g) = [x \to (f(x) - g(x)) \cap [\![0;+\infty[\![\,]$,
- $\text{SYNC}_{\mathscr{I}}(t,f) = [x \to f(x) \cap [\![t(x);+\infty[\![]$,
- $0_{\mathscr{I}} = [x \to [\![0;0]\!]\,]$,
- $1_{\mathscr{I}}(v) = \left[\begin{cases} x \to [\![1;1]\!] & \text{if } x = v \\ x \to [\![0;0]\!] & \text{otherwise} \end{cases}\right].$

### 9.3.2 Linear equalities domain

The complete lattice $(\mathscr{K}, \sqsubseteq_{\mathscr{K}}, \sqcup_{\mathscr{K}}, \top_{\mathscr{K}}, \sqcap_{\mathscr{K}}, \bot_{\mathscr{K}})$ of linear equality systems between the finite set of variables $\mathscr{V}_c$ is described with its lattice operations in [47]. This domain uses Gaussian elimination in order to normalize systems. It is related to the domain $\wp(\mathbb{N}^{\mathscr{V}_c})$ via the monotonic function $\gamma_{\mathscr{K}}$ which maps each system to the set of its solutions. Moreover, since there is no infinite ascending chain [47], we can choose $\sqcup_{\mathscr{K}}$ as a widening operator. To define the addition $+_{\mathscr{K}}$ and the subtraction $-_{\mathscr{K}}$ of two systems, we compute a particular solution of each system

$O_1$ and $O_2$, and a linear direction $\overrightarrow{H_1}$ and $\overrightarrow{H_2}$. Then we use the following equalities:

$$\begin{aligned}(O_1+\overrightarrow{H_1})+_{\mathscr{K}}(O_2+\overrightarrow{H_2}) &= (O_1+^{\sharp}O_2)+(\overrightarrow{H_1}\sqcup_{\mathscr{K}}\overrightarrow{H_2}),\\ (O_1+\overrightarrow{H_1})-_{\mathscr{K}}(O_2+\overrightarrow{H_2}) &= (O_1-^{\sharp}O_2)+(\overrightarrow{H_1}\sqcup_{\mathscr{K}}\overrightarrow{H_2}).\end{aligned}$$

Such a decomposition may be extracted from the normal form in linear time, so the cost of affine addition and subtraction is cubic due to the use of the Gaussian elimination.

Other primitives are defined as follows:

- $\textsc{sync}_{\mathscr{K}}(I,K)=K$,
- $0_{\mathscr{K}}=\{x=0,\ \forall x\in\mathscr{V}_c$,
- $1_{\mathscr{K}}(v)=\begin{cases} x=1 & \text{if } x=v\\ x=0 & \text{otherwise.}\end{cases}$

Roughly speaking, synchronization cannot be directly checked in $\mathscr{K}$. So we define it as the identity. Linear constraints will therefore be used to prove that synchronization interval constraints are incompatible, by reduction. Other primitive definitions are straightforward.

### 9.3.3 Approximated reduced product

Our numerical domain is the product $(\mathscr{I}\times\mathscr{K})$. It is partially ordered by the pair-wise ordering $\sqsubseteq_{\mathscr{I}}\times\sqsubseteq_{\mathscr{K}}$. It is related to $\wp(\mathbb{N}^{\mathscr{V}_c})$ by the concretization function $\gamma_{\mathscr{N}_{\mathscr{V}_c}}$ where $\gamma_{\mathscr{N}_{\mathscr{V}_c}}(i,s)$ is defined as $\gamma_{\mathscr{I}}(i)\cap\gamma_{\mathscr{K}}(s)$. Generic primitives are expressed as follows:

- $\bot_{\mathscr{N}_{\mathscr{V}_c}}$, $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}$, $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$, $+^{\sharp}$, $\sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}$, $0_{\mathscr{N}_{\mathscr{V}_c}}$, $1_{\mathscr{N}_{\mathscr{V}_c}}$ are defined pair-wisely,
- $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}(A,(i,s))=\rho(i',s)$,

  where $\begin{cases} i'(x)=i(x)\cap[\![1;+\infty|[ & \forall x\in A\\ i'(x)=i(x) & \forall x\in\mathscr{L}_{\mathrm{p}}\setminus A,\end{cases}$

  and $\rho$ satisfies $\forall a\in\mathscr{I}\times\mathscr{K},\ \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a)\sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}\gamma_{\mathscr{N}_{\mathscr{V}_c}}(\rho(a))$.

The definition of $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}$ uses a reduction operator $\rho$. Roughly speaking, the operator $\rho$ simplifies constraints without losing any solution. We now present a reduction $\rho$ between $\mathscr{I}$ and $\mathscr{K}$. It consists in taking into account linear constraints in order to narrow interval ranges. For instance, the system of constraints $x+y=12 \wedge x \in [\![3;15]\!] \wedge y \in [\![4;19]\!]$ can be reduced to the system $x+y=12 \wedge x \in [\![3;8]\!] \wedge y \in [\![4;9]\!]$. Linear constraints are likely to be combined, via Gaussian elimination, in order to give new linear constraints which will allow for further reductions. Therefore, generating the whole set of such combinations is likely to require an exponential time of execution.

We propose a two-step polynomial algorithm for computing an approximate solution to this problem. The first step aims at narrowing infinite intervals into finite ones. It uses Gaussian elimination to obtain a positive representation of systems of linear equalities, that is to say, an equivalent system of equations such that if a variable occurs with a strictly negative coefficient in an equation, then this variable occurs with a negative coefficient in each equation. Positive representations contain only a few undefined forms[1], which allows narrowing most of the infinite intervals into finite ones with a $\mathcal{O}(|\mathcal{V}_c|^3)$ worst-case. The second step is inspired by [20]: it consists in obtaining a triangular system of constraints of the form $a_1.x_1+\ldots+a_n.x_n \in I$ where $I$ is an interval. This system is then used to propagate unidirectionally intervals from non-diagonal to diagonal variables. The result is a good reduction with a $\mathcal{O}(|\mathcal{V}_c|^3)$ worst-case. In the case that the algorithm discovers an unsatisfiable constrain, the result of the reduction is set to $\perp_{\mathcal{N}_{\mathcal{V}_c}}$.

### 9.3.3.1 Solving undefined forms

Let $\mathcal{V}_c^{\text{inf}}$ be a subset of $\mathcal{V}_c$ and $K$ a finite system of linear constraints on the variables $\mathcal{V}_c$. We denote by $K$ the system of equations:

$$\left\{\sum\{a_i^v.v \mid v \in \mathcal{V}_c\} = b_i, \forall i \in [\![1;n]\!]\right. .$$

We first define a positive form with respect to $\mathcal{V}_c^{\text{inf}}$ as follows:

**Definition 9.3.1.** $K$ is in positive form with respect to $\mathcal{V}_c^{\text{inf}}$ if and only if for any $v \in \mathcal{V}_c$, if there exist $i_1, i_2 \in [\![1;n]\!]$ such that $a_{i_1}^v < 0 < a_{i_2}^v$, then $v \notin \mathcal{V}_c^{\text{inf}}$.

This way, the variables which may occur in the matrix describing $K$ with both a positive and a negative coefficient are known to be bounded[2]. Such a form can be computed by using Gaussian elimination with a $\mathcal{O}(|\mathcal{V}_c|^3)$ worst-case time cost. A positive form contains only few undefined forms. For each constraint in which all variables with an infinite range occur with the same sign, we can narrow the range of variables that occurs in this constraint into finite intervals. A dynamic resolution of such a system leads to a reduction step in $\mathcal{O}(|\mathcal{V}_c|^2)$.

[1] An undefi ned form is a subtraction between two unbounded intervals.

[2] $\mathcal{V}_c^{\text{inf}}$ denotes the set of the variables which are not proved to be bounded.

**Example 9.3.2.** *We consider the set of variables $\mathscr{V}_c = \{x_i \mid 1 \leq i \leq 9\}$. We suppose that these variables are totally ordered by the order $\leq_{\mathscr{V}_c}$ that is defined as: $x_i \leq_{\mathscr{V}_c} x_j$ if and only if $i \leq j$. We also suppose that we have no interval constraints yet, so we set $\mathscr{V}_c^{\inf} = \mathscr{V}_c$. We consider the following system (in normal form (see [47])):*

$$\begin{cases} x_1 - x_7 + x_8 + x_9 = 0 & (L_1^1) \\ x_2 - x_7 + x_8 + x_9 = 0 & (L_2^1) \\ x_3 + x_4 + x_5 + x_6 + 2.x_7 - 2.x_8 - 2.x_9 = 3 & (L_3^1) \end{cases}$$

*The variable $x_7$ occurs with a positive coefficient in $(L_3^1)$ and with a negative coefficient in $(L_2^1)$. So, we replace the constraint $L_1^1$ with the affine combination of $(L_1^1)$ (with the coefficient $1$) and of $(L_3^1)$ (with the coefficient $\frac{1}{2}$); we also replace the constraint $L_2^1$ with the affine combination of $(L_2^1)$ (with the coefficient $1$) and of $(L_3^1)$ (with the coefficient $\frac{1}{2}$). Thus, we get the system:*

$$\begin{cases} x_1 + \frac{1}{2}.x_3 + \frac{1}{2}.x_4 + \frac{1}{2}.x_5 + \frac{1}{2}.x_6 = \frac{3}{2} & (L_1^2) \\ x_2 + \frac{1}{2}.x_3 + \frac{1}{2}.x_4 + \frac{1}{2}.x_5 + \frac{1}{2}.x_6 = \frac{3}{2} & (L_2^2) \\ x_3 + x_4 + x_5 + x_6 + 2.x_7 - 2.x_8 - 2.x_9 = 3 & (L_3^2) \end{cases}$$

*This system is in positive form. Moreover, we can deduce that the variables $x_1$ and $x_2$ range in the interval $[\![0;1]\!]$ and that the variables $x_3$, $x_4$, $x_5$, and $x_6$ range in the interval $[\![0;3]\!]$.*

*Remark* 9.3.3. This Gaussian elimination only involve linear combinations with positive coefficients. So, a variable that never occurs with a negative coefficient before the elimination step, never occurs with a negative coefficient after the elimination step.

#### 9.3.3.2 Narrowing finite intervals

The second step is inspired by [20]: it consists in obtaining a triangular system of constraints of the form $a_1.x_1 + \ldots + a_n.x_n \in I$ where $I$ is an interval. This system is then used to propagate unidirectionally intervals from non-diagonal to diagonal variables. The result is a good reduction with a $\mathscr{O}(|V|^3)$ worst-case time cost.

We use three kinds of reductions:

1. Gaussian elimination:

$$\begin{cases} x + y + z = 1 \\ x + y + t = 2 \end{cases} \implies \begin{cases} x + y + z = 1 \\ t - z = 1, \end{cases}$$

2. interval propagation:

$$\begin{cases} x+y+z=3 \\ x \in [\![0;5]\!] \\ y \in [\![0;6]\!] \\ z \in [\![0;8]\!] \end{cases} \Longrightarrow \begin{cases} x+y+z=3 \\ x \in [\![0;\mathbf{3}]\!] \\ y \in [\![0;6]\!] \\ z \in [\![0;8]\!], \end{cases}$$

3. redundancy introduction:

$$\begin{cases} x+y-z=3 \\ x \in [\![1;2|[ \end{cases} \Longrightarrow \begin{cases} x+y-z=3 \\ y-z \in [\![1;2]\!] \\ x \in [\![1;2]\!]. \end{cases}$$

First, we use Gaussian elimination to get a normal form of the linear constraint system, we then use interval propagation to narrow the range of the interval of the pivot of each constraint. Then, we forget about the pivot using redundancy introduction. Then, we get a new system involving only variables which were not a pivot of a constraint in the previous one. Then, we proceed recursively with it until constraints contain some variables. Then, we consider all the constraints we have computed which form a triangular system. We propagate the information we have collected on the interval ranges backward, by applying interval propagation from non-diagonal variables to diagonal ones, starting from the last constraint and ending with the first one.

## 9.4 Prototypes and analysis examples

### 9.4.1 Two prototypes

We now present some examples of analysis results. These results have been automatically computed by using the $\pi$-s.a prototype [34] for systems that are written in the $\pi$-calculus, and the *amb*-s.a. prototype [33] for systems that are written in the *ambient* calculus. These two prototypes belong to preliminary works, since they do not work on the meta language level. They propose the choice between several numerical domains: the intervals [22], the product between the intervals and the affine equalities [47], the octagons [55,58], and the abstract multi-sets [43] (based on the use of disjunctive completion). In the case of the $\pi$-calculus, time complexity results are given in [32]. These bounds are a bit too pessimistic, since we bounded the reduction time complexity by $\mathcal{O}(|\mathcal{V}_c|^4)$ instead of $\mathcal{O}(|\mathcal{V}_c|^3)$, where $n$ is the number of variables.

### 9.4.2 Examples

We now describe results obtained on our examples. For the sake of brevity and simplicity, we do not present linear constraints. They are not of interest when considering the result as they are used internally to refine the interval information. Interval constraints are tagged with the actions of syntactic threads.

**Example 9.4.1.** *We give in Fig. 9.4 the result of occurrence analysis of the* ftp-server *(Cf. Ex. 2.1.1 on page 18). This result ensures that no more than three syntactic instances of the syntactic thread* $\boldsymbol{deal}![\boldsymbol{data}]$ *may simultaneously occur. So we can conclude that no more than three sessions may be active at the same time. This constraint is proved using the linear constraint which proves that the sum of the numbers of available ports* $\mathbf{port}![]$ *and of activated sessions* $\boldsymbol{deal}![\boldsymbol{data}]$ *is always equal to three.*

**Example 9.4.2.** *We give in Fig. 9.5 the occurrence analysis result for the* token-ring *(Cf. Ex. 2.1.2 in page 18). This result ensures that only one syntactic instance of the thread* $\boldsymbol{crit}![]$ *may occur at any time. So only one process may proceed its critical section at the same time.*

**Remark 9.4.3.** *The analysis may not succeed in proving that the occurrence number of the syntactic thread* $\mathbf{mon}![\boldsymbol{left}, \mathbf{left0}]$ *is less than or equal to* 1 *without counting the number of performed transitions. This is because it is an asynchronous thread. On the other hand, only a linear number of extra variables are required to prove this property.*

**Example 9.4.4.** *We enrich Ex. 3.1.1 (Cf. Ex. 3.1.1 on page 38) with some tokens in order to ensure that the server may not launch more than three simultaneous instances. The enriched system is given in Fig. 9.6. The ambients named* **token** *denotes some tokens. They may be consumed when they are enclosed in the server* **server**. *When consumed, they launch a server instance. Once the session is released, a new ambient named* **token** *is launched which models that another server instance is available. It is worth noting that this* **token** *ambient contains explicit rooting information so that the token may leave the packet and enter the scope of the* **server** *ambient.*

*Occurrence counting results are also given in Fig. 9.6. The analysis automatically discovers that the threads that are launched inside an* **instance** *ambient may not occurs more than three times in the whole system. It is worth noting that the analysis fails in proving that there may not be more than three occurrences of the* **instance** *ambient simultaneously. This is because the lack of affine invariant that relates the number of ambients named* **token** *and the number of ambients named* **instance**. *Nevertheless, this information can be restored by applying a reduced*

$$
\begin{array}{l}
((\nu\ \mathrm{make})(\nu\ \mathrm{server})(\nu\ \mathrm{port})\\
((*\mathrm{make}?^{1}[\,](\nu\ \mathit{address})(\nu\ \mathit{request})\\
\qquad\qquad (\\
\qquad\qquad\ (*\mathit{address}?^{\llbracket 0;+\infty\llbracket}[\,]\mathrm{server}!^{\llbracket 0;+\infty\llbracket}[\mathit{address},\mathit{request}])\\
\qquad\qquad |\\
\qquad\qquad\ \mathit{address}!^{\llbracket 0;+\infty\llbracket}[\,]\\
\qquad\qquad |\\
\qquad\qquad\ \mathrm{make}!^{\llbracket 0;1\rrbracket}[\,]\\
\qquad\qquad ))\\
|\\
\ (*\mathrm{server}?^{1}[\mathit{email},\mathit{data}](\nu\ \mathit{deal})\\
\qquad\qquad (\\
\qquad\qquad\ \mathrm{port}?[\,]^{\llbracket 0;+\infty\llbracket}\\
\qquad\qquad\qquad (\\
\qquad\qquad\qquad\ \mathit{deal}!^{\llbracket 0;3\rrbracket}[\mathit{data}]\\
\qquad\qquad\qquad |\\
\qquad\qquad\qquad\ \mathit{deal}?^{\llbracket 0;3\rrbracket}[\mathit{rep}](\mathit{email}!^{\llbracket 0;+\infty\llbracket}[\mathit{rep}]\mid \mathrm{port}!^{\llbracket 0;3\rrbracket}[\,])\\
\qquad\qquad\qquad )\\
\qquad\qquad \oplus\\
\qquad\qquad\ \mathit{email}!^{\llbracket 0;+\infty\llbracket}[\,]\\
\qquad\qquad ))\\
\mid \mathrm{port}!^{\llbracket 0;1\rrbracket}[\,]\\
\mid \mathrm{port}!^{\llbracket 0;1\rrbracket}[\,]\\
\mid \mathrm{port}!^{\llbracket 0;1\rrbracket}[\,]\\
\mid \mathrm{make}!^{\llbracket 0;1\rrbracket}[\,])\\
)
\end{array}
$$

Figure 9.4: The *ftp*-server occurrence analysis.

$$\begin{array}{l}
((\nu\ \mathrm{make})(\nu\ \mathrm{mon})(\nu\ \mathrm{left0}) \\
( \\
\quad (\ (*\mathrm{make}?^{1}[\mathit{left}](\nu\ \mathit{right})(\mathrm{mon}!^{\llbracket 0;+\infty\llbracket}[\mathit{left},\mathit{right}]\ |\ \mathrm{make}!^{\llbracket 0;1\rrbracket}[\mathit{right}])) \\
\quad |\ (*\mathrm{make}?^{1}[\mathit{left}](\mathrm{mon}!^{\llbracket 0;1\rrbracket}[\mathit{left},\mathrm{left0}])) \\
\quad |\ \mathrm{make}!^{\llbracket 0;1\rrbracket}[\mathrm{left0}]) \\
| \\
\quad ((*\mathrm{mon}?^{1}[\mathit{prev},\mathit{next}] \\
\qquad\qquad (*\mathit{prev}?^{\llbracket 0;+\infty\llbracket}[](\nu\ \mathit{crit})(\mathit{crit}?^{\llbracket 0;1\rrbracket}[]\mathit{next}!^{\llbracket 0;1\rrbracket}[] \\
\qquad\qquad\qquad\qquad\qquad\qquad |\ \mathit{crit}!^{\llbracket 0;1\rrbracket}[]))) \\
\quad |\ \mathrm{left0}!^{\llbracket 0;1\rrbracket}[])))
\end{array}$$

Figure 9.5: The token-ring occurrence analysis.

*product with the information that are derived by the analysis that is proposed in Chap. 10.*

### 9.4.3 Implementation issue

It is usually very difficult to scale up when using relational domains. Even a domain which is quadratic in time may not scale up.

In order to speed up the analysis, we describe affine system by block-wise matrices. At the beginning, matrices are diagonal. During computation we merge blocks as soon as they are involved in the same computation step. The advantage is that all affine operations (including the reduction) can be performed block-wise, which decrease their complexity (which is the sum of the cube of the size of the blocks instead of the cube of the sum of the size of the blocks).

The two following approaches have not been investigated yet. The first one one is variable merging: instead of distinguishing the variables $x$, $y$, and $z$, we may be interested in the sum $x+y+z$. This allows for considering fewer variables; complexity can also be mastered by a packing strategy [8,57,53]: packing consists in computing some packs of variables and in only detecting relationships between variables in the same packs, these packs can be defined by the end-user, or they can be computed during a pre-analysis. It would be very interesting to design some strategy either to merge some variables or to define some packs, in the case of our occurrence counting analysis.

```
$(\nu\ \text{request})(\nu\ \text{make})(\nu\ \text{server})(\nu\ \text{duplicate})(\nu\ \text{instance})$
$(\nu\ \text{answer})(\nu\ \text{client})(\nu\ \text{token})$
$($
 $\text{server}^1[$
            $!open^1\text{duplicate}.0$
            $|$
            $!(k)^1.open^{\llbracket 0;+\infty\llbracket}\text{token}.$
                 $\text{instance}^{\llbracket 0;+\infty\llbracket}[$
                                  $in^{\llbracket 0;3\rrbracket}k.open^{\llbracket 0;3\rrbracket}\text{request}.(rep)^{\llbracket 0;3\rrbracket}.$
                                     $($
                                      $\text{answer}^{\llbracket 0;+\infty\llbracket}[<rep>^{\llbracket 0;+\infty\llbracket}]$
                                     $|$
                                      $out^{\llbracket 0;3\rrbracket}\text{server}.\text{token}^{\llbracket 0;3\rrbracket}[out^{\llbracket 0;+\infty\llbracket}k.in^{\llbracket 0;+\infty\llbracket}\text{server}.0]$
                                     $)$
                                $]$
            $|$
            $\text{token}^{\llbracket 0;1\rrbracket}[0]\ |\ \text{token}^{\llbracket 0;1\rrbracket}[0]\ |\ \text{token}^{\llbracket 0;1\rrbracket}[0]$
            $]$
$|$
 $\text{client}^1[!(x)^1.((\nu\ q)(\nu\ p)$
                     $p^{\llbracket 0;+\infty\llbracket}[$
                                $out^{\llbracket 0;+\infty\llbracket}\text{client}.0$
                                $|$
                                $\text{request}^{\llbracket 0;+\infty\llbracket}[<q>^{\llbracket 0;+\infty\llbracket}\ [<q>]$
                                $|$
                                $open^{\llbracket 0;+\infty\llbracket}\text{instance}.0$
                                $|$
                                $in^{\llbracket 0;+\infty\llbracket}\text{server}.\text{duplicate}^{\llbracket 0;+\infty\llbracket}[out^{\llbracket 0;+\infty\llbracket}p.<p>^{\llbracket 0;+\infty\llbracket}]$
                                $]$
                    $|$
                     $<\text{make}>^{\llbracket 0;1\rrbracket}$
                    $)$
           $|$
            $<\text{make}>^{\llbracket 0;1\rrbracket}$
           $]$
$)$
```

Figure 9.6: The ambient *ftp*-server analysis.

# Chapter 10

# Thread partitioning

We have proposed in Chap. 8 a first analysis that focus on the potential links between threads. This first analysis captures dynamic aspects of systems, since it distinguishes among the distinct instances of syntactic objects. Unfortunately, this first analysis abstracts away concurrency, since it abstracts each thread separately. It forgets away that some threads may not occur simultaneously. On the other side, we have proposed in Chap. 9 an analysis that focuses on the concurrency properties. This second analysis captures which threads may occur simultaneously. But this second analysis abstracts away the potential links between threads. A Cartesian product (Cf. Sect. 7.3.1 on page 160) between these two analyses gives accurate results. It implicitly uses the coalescent product: a global interaction is enabled in the product analysis, only if it is enabled in each analysis. Unfortunately, that is the only reduction that is performed.

In this chapter, we propose an analysis that captures both dynamic and concurrency properties. The state of a system is a set of threads. We propose to partition the set of threads according to dynamical information (such as the value of a given variable). Then, for each class of the partition, we count the number of thread instances in this class. This idea comes from analyses for the *ambient* calculus. It is quite easy to abstract the content of each ambient. Furthermore, each ambient denotes a computation unit, so it is relevant to abstract the content of each ambient separately. Our goal is then to generalize this approach to models where the notion of computation unit is not explicit. Thus, we specify as a parameter what a computation unit is. Then, the analysis abstracts the threads in each computation unit separately.

In Sect. 10.1, we give some examples that illustrate the kind of properties we are interested in. In Sect. 10.2, we introduce an analysis for the *ambient* calculus. This analysis captures a description of the content of each ambient. We generalize this approach in Sect. 10.3 and give some examples of analysis results in Sect. 10.4.

## 10.1 Motivating examples

### 10.1.1 Shared-memory with dynamical allocation

**Example 10.1.1.** *A shared memory with dynamic allocation of memory cells can be described in the $\pi$-calculus by the system given in Fig. 10.1. We use several font styles to distinguish several kinds channel names, and several kind variables. Global channel names are written in* roman. *Local variables are written in italic. There are two kinds of names associated with memory cells: some names have an internal use, they are written in* `typewriter font`*; some names denote capability to control a memory cell, they are written in* SMALL CAPITAL LETTERS. *The names that describe the data of clients are written in* ***bold****. The initial unsafe names are* <u>*underlined*</u>.

*The state of each memory cell is described by using three names. The name* `cell` *encodes the content of the cell. The content of the cell is supposed to be output on the channel named* `cell`*. Moreover, a cell may be locked or unlocked, which is described by two names* `mutex` *and* `nomutex`*. If the signal* `mutex`![] *occurs and the signal* `nomutex` *does not occur, then the cell is unlocked; if the signal* `nomutex`![] *occurs and the signal* `mutex` *does not occur, then the cell is locked. Interactions with a memory cell are described by using four names. The name* READ *provides the capability to read the content of the cell. The name* WRITE *provides the capability to overwrite the content of the cell. The name* LOCK *provides the capability to lock the cell. The name* UNLOCK *provides the capability to unlock the cell.*

*The global name* create *allows the creation of new memory cells. The global name* null *describes the content of memory cells at initialization. Whenever a message is sent via the channel named* create*, an instance of memory cell is launched:*

1. *seven fresh names (*`cell`*,* `mutex`*,* `nomutex`*,* READ*,* WRITE*,* LOCK *and* UNLOCK*) are declared;*

2. *the content of the cell is initialized to the global name* null*, the cell is declared unlocked (*`nomutex`![]*);*

3. *the four capability names are returned at an address given at the creation of the cell: this gives the client the capability to interact with its memory cell;*

4. *four resources are spawn; each of them describes the behavior of the cell when it interacts with a client:*

   - *a read operation requires one argument that denotes a return address: the content of the cell is read and returned at the given address (please*

```
$(\nu^a \text{create})(\nu^b \text{null})$
(
 $*\text{create}?^1[d]$
   $(\nu^c \texttt{cell})(\nu^d \texttt{mutex})(\nu^e \texttt{nomutex})$
   $(\nu^f \textsc{write})(\nu^g \textsc{read})(\nu^h \textsc{lock})(\nu^i \textsc{unlock})$
               (
                $\texttt{cell}!^2[\text{null}]$
               |
                $\texttt{mutex}!^3[]$
               |
                $d!^4[\textsc{read},\textsc{write},\textsc{lock},\textsc{unlock}]$
               |
                $*\textsc{read}?^5[port].cell?^6[u].(\texttt{cell}!^7[u] \mid port!^8[u])$
               |
                $*\textsc{write}?^9[u,ack].\texttt{cell}?^{10}[v](\texttt{cell}!^{11}[u] \mid ack!^{11}[])$
               |
                $*\textsc{lock}?^{12}[ack].\texttt{mutex}?^{13}[](ack!^{14}[] \mid \texttt{nomutex}!^{15}[])$
               |
                $*\textsc{unlock}?^{16}[ack].\texttt{nomutex}?^{17}[](ack!^{18}[] \mid \texttt{mutex}!^{19}[])$
               )
|
 $*\underline{rec}^{20}?[].$
   $(\nu^j \boldsymbol{data})(\nu^k \boldsymbol{add_1})(\nu^l \boldsymbol{add_2})(\nu^m \boldsymbol{trace})(\nu^n \boldsymbol{ack_1})(\nu^o \boldsymbol{ack_2})(\nu^p \boldsymbol{ack_3})$
       $\text{create}!^{21}[\boldsymbol{add_1}].\boldsymbol{add_1}?^{22}[r,w,m,n].m!^{23}[\boldsymbol{ack_1}].\boldsymbol{ack_1}?^{24}[].w!^{25}[\boldsymbol{data},\boldsymbol{ack_2}].$
       $\boldsymbol{ack_2}?^{26}[].r!^{27}[\boldsymbol{add_2}]n!^{28}[\boldsymbol{ack_3}].\boldsymbol{ack_3}?^{29}[].\boldsymbol{add_2}?^{30}?[d].\boldsymbol{trace}!^{31}[d]$
|
 $*\underline{context}?^{32}[d].\text{create}!^{33}[d]$
)
```

Figure 10.1: A shared memory with dynamic allocation in the $\pi$-calculus.

*note that we copy the content of the cell once, not to loose it);*

- *a write operation requires two arguments: the new content and an acknowledgment address: first, the content of the cell is removed; then, it is replaced by the new content, while a signal is sent at the acknowledgment address; acknowledgments control the sequentiality of client requests;*
- *a lock operation requires one argument that denotes an acknowledgment address: the operation may be performed as soon the mutex is available, the mutex is taken and the acknowledgment is sent;*
- *an unlock operation requires one argument that denotes an acknowledgment address: the operation may be performed as soon as the mutex is not available, the mutex is released and the acknowledgment is sent.*

*Mutex policy is left at the charge of the client.*

*Then we consider an unbounded number of clients. Each client is described by two return addresses* $\mathbf{add_1}$ *and* $\mathbf{add_2}$*, three acknowledgment addresses* $\mathbf{ack_1}$*,* $\mathbf{ack_2}$ *and* $\mathbf{ack_3}$*, a data* ***data****, and another name* ***trace*** *that is used to keep a trace of the client session. Each client is created by the context (which uses the unsafe name* $\underline{\text{rec}}$*). Each client creates a cell and receives on the address* $\mathbf{add_1}$ *the capabilities to interact with this cell. Then, the client locks the cell and waits for the acknowledgment. Then, the client writes its data and waits for the acknowledgment. Then, the client unlocks the cell and waits for the acknowledgment. Then, the client reads the content of the cell and spawns a trace of the session.*

*The context may also create memory cells, by sending message on the unsafe name* $\underline{\text{context}}$*. The analysis given in Sect. 8.2.11 on page 213 proves that the client data are not sent neither to other clients, nor to the context. But it is not able to capture the concurrency aspects of this system. We would like to prove that for each memory cell:*

- *there is always at most one emission over the channel named* `cell`*;*
- *there is always at most one signal* `mutex` *or* `nomutex`*.*

*Occurrence counting analysis (Cf. Chap. 9) may not help because it merges all distinct instances of memory cells. The analysis proposed in this chapter succeeds in proving those properties.*

### 10.1.2 Authentication in protocol

**Example 10.1.2.** *We consider in Fig. 10.2 a simplified version of the Woo and Lam one-way public-key authentication protocol [76], in which host names are*

$$
\begin{array}{l}
(\nu^{\alpha} sk_A)(\nu^{\beta} sk_B) \\
\qquad \overline{c}^{1}\langle \mathrm{pk}^{a}(sk_A)\rangle.\overline{c}^{2}\langle \mathrm{pk}^{b}(sk_B)\rangle. \\
\qquad\qquad (\\
\qquad\qquad\ !c^{3}\langle x\_pk_B\rangle.\mathrm{begin}^{4}(x\_pk_B).\overline{c}^{5}\langle \mathrm{pk}^{c}(sk_A)\rangle.c^{6}\langle x\_b\rangle. \\
\qquad\qquad\quad \overline{c}^{7}\langle \mathrm{sign}^{d}(\mathrm{tuple}_{n}^{e}(\mathrm{pk}^{f}(sk_A), x\_pk_B, x\_b), sk_A)\rangle \\
\qquad\qquad | \\
\qquad\qquad\ !c^{8}\langle x\_pk_A\rangle.(\nu^{\gamma} b) \\
\qquad\qquad\quad \overline{c}^{9}\langle b\rangle.c^{10}\langle m\rangle. \\
\qquad\qquad\qquad \mathrm{let}^{11} x = \mathrm{checksign}(m, x\_pk_A) \text{ in} \\
\qquad\qquad\qquad \mathrm{let}^{12} y\_pk_A = \mathrm{th}_{1}^{3}(x) \text{ in} \\
\qquad\qquad\qquad \mathrm{let}^{13} y\_pk_B = \mathrm{th}_{2}^{3}(x) \text{ in} \\
\qquad\qquad\qquad \mathrm{let}^{14} y\_b = \mathrm{th}_{3}^{3}(x) \text{ in} \\
\qquad\qquad\qquad [y\_pk_A = x\_pk_A]^{15}. \\
\qquad\qquad\qquad [y\_pk_B = \mathrm{pk}^{g}(sk_B)]^{16}. \\
\qquad\qquad\qquad [y\_b = b]^{17}. \\
\qquad\qquad\qquad [x\_pk_A = \mathrm{pk}^{h}(sk_A)]^{18}. \\
\qquad\qquad\qquad \mathrm{end}^{19}(\mathrm{pk}^{i}(sk_B)) \\
\qquad\qquad )
\end{array}
$$

Figure 10.2: Woo and Lam one-way public-key authentication protocol.

*replaced with public keys. We consider two principals A and B. The channel c is unsafe: The context may send and listen on it. The system begins with the creation of two secret keys for A and B. The public version of these keys are sent on the unsafe channel. Both principals A and B may run an unbounded number of sessions. At each session, the principal A first receives a message containing the public key of its interlocutor. Then, the principal A emits a signal* begin *at program point* 4 *with the public key of its interlocutor: this means that the principal A has started a session with the host of the public key that it has received. Then, the principal A sends its public key. Then, it waits for a message that should contain a nonce created by its interlocutor. Then, it sends a tuple that contains its public key, the public key of its interlocutor and the nonce. This tuple is signed with its secret key. First, the principal B receives a message containing the public key of its interlocutor. Then, the principal B creates a nonce. It sends this nonce and waits for a message. If this message is a tuple that contains the public key of its interlocutor, its own public key, and the nonce and if this message is signed by the secret key of it interlocutor, the principal B concludes that this message has been created by the principal A. It sends the signal* end *with the public key of B.*

*A context may not sign messages with the secret key of A. A context may only force a session between the principal B by receiving the signed message from the principal A, then by sending this message to the principal B. The nonce is used to prevent the system to use the same spied message several times.*

*The analysis proposed in this chapter succeeds in proving that any signal* end$(x)$ *follows a signal* begin$(x)$ *with the same argument x. This is the* non-injective agreement *property. We do not prove the* injective-agreement *property yet, that is ensured by the use of the nonce. We leave this last property in future works. Blanchet proposes in [7] an analysis that proves the* injective-agreement *property for this example.*

## 10.2 Analyzing the content of an ambient

In this section, we describe an analysis that captures an approximation of the content of each ambient. More precisely, we want to relate each label $l \in \mathscr{L}$ to an abstraction of the content of any ambient labeled $l$. Nielson *et al.* have already proposed such an analysis in [63]. In this analysis, ambients are partitioned into an exponential (with respect to the initial state program point number) number of equivalence classes. Moreover, the content of ambients in each equivalence class is described in a domain the height of which is also exponential. So the worst case cost in both time and space, is at least a double exponential with respect to the number of program points in the initial state of the system. In our analysis, we consider only a quadratic number of classes (with respect to the number of program points) and we use the reduced product between the interval and the affine equalities (Cf. Chap. 9) to describe the content of the ambients in each equivalence class. Thus we get a polynomial analysis $(\mathscr{O}(n^{i*3+4}))$, where $n$ is the number of program points in the initial state and $i$ is a parameter chosen among 1, 2, and 3.

### 10.2.1 Abstract domain

We consider the non standard semantics for the *ambient* calculus that we have described in Chap. 3.2. We want to relate each ambient with an abstraction of its content. More precisely, we will abstract the occurrence number of threads inside each ambient. Since, there is an unbounded number of potential ambients, we gather the description of the content of any ambient that have been created at the same program point (i.e. we merge the abstraction of the ambient $(i, id_i)$ and of the ambient $(i, id_j)$). Intuitively, our abstract domain could be the set $\mathscr{L} \rightarrow \mathbb{N}_{\mathscr{L}}$ of the function mapping each label $l$ into a vector which indicates the number of threads at each program points that may be located in the same ambient labeled $l$ at the same time. However, this domain is not rich enough:

- the content of an ambient $a$ is often relating to the location of the ambient $a$;

- introducing extra variables (see Chap. 9) helps in expressing some invariants (since it allows the description of affine inequalities by using only affine equalities).

Our main domain handles extra variables. Moreover, we partition the abstraction of each ambient content according to the location of this ambient.

#### 10.2.1.1 Notations

We recall that $\mathscr{L}$ is the set of labels. Labels in $\mathscr{L}$ are used to locate the syntactic components of the system $\mathscr{S}$. We suppose that the set $\mathscr{L}$ is partitioned into three subsets of labels $\mathscr{L}_{\mathrm{p}}$, $\mathscr{L}_{amb}$ and $\mathscr{L}_c$, such that labels in $\mathscr{L}_{\mathrm{p}}$ tag capabilities and input/output actions, labels in $\mathscr{L}_{amb}$ tag ambient creations, and labels in $\mathscr{L}_c$ tag name restrictions.

For any program point $P$, we denote by $lab(P)$ the label of the program point, by $act(P)$ the action that is associated to the program point, and by $cont(P)$ the continuation that is associated to the program point. Thus, we define:

$$lab = \begin{cases} n^l[\bullet] & = l, \\ in^l\ n.P & = l, \\ out^l\ n.P & = l, \\ open^l\ n.P & = l, \\ !open^l\ n.P & = l, \\ (n)^l.P & = l, \\ !(n)^l.P & = l, \\ \langle n \rangle^l & = l; \end{cases} \qquad act = \begin{cases} n^l[\bullet] & = ambient, \\ in^l\ n.P & = in, \\ out^l\ n.P & = out, \\ open^l\ n.P & = open, \\ !open^l\ n.P & = !open, \\ (n)^l.P & = input, \\ !(n)^l.P & = !input, \\ \langle n \rangle^l & = output; \end{cases}$$

$$cont = \begin{cases} n^l[\bullet] & = 0, \\ in^l\ n.P & = P, \\ out^l\ n.P & = P, \\ open^l\ n.P & = P, \\ !open^l\ n.P & = P, \\ (n)^l.P & = P, \\ !(n)^l.P & = P, \\ \langle n \rangle^l & = 0. \end{cases}$$

#### 10.2.1.2 Partitioning

We partition set of threads according to the computation unit of these threads. A computation unit is given by two ambient identities $loc \in \mathscr{L}_{amb} \times \mathscr{M}$ and $loc' \in \mathscr{L}_{amb} \times \mathscr{M}$, and is written $loc^{loc'}$. The computation unit $loc^{loc'}$ denotes the potential content of the ambient $loc$ when the ambient $loc$ is inside the ambient $loc'$. The set of all computation units is denoted by *unit*. The top level ambient ($(\text{top},\varepsilon)$) never migrates. Moreover the top level ambient is surrounded by no ambient. Thus, we introduce an extra computation unit $(\text{top},\varepsilon)^{(\text{top},\varepsilon)}$ that denotes the content of the top level ambient.

Let $C \in \mathscr{C}$ be a concrete configuration (we recall that $\mathscr{C}$ is the set of the reachable configuration). For each thread $t = (P, id, loc, E) \in C$, we define the program point $give\text{-}pp(t)$ of the thread $t$ as the label $lab(P)$, the identity of the thread $give\text{-}id(t)$ as the thread identity $(lab(P), id)$, the location of the thread $give\text{-}father(t)$ as the thread identity $loc$. Then, we define the ambient $give\text{-}big\text{-}father_C(t)$ that encloses the surrounding ambient as the unique thread identity that satisfies:

1. $give\text{-}big\text{-}father_C(t) = (\text{top},\varepsilon)$, if $loc = (\text{top},\varepsilon)$;

2. $\mathit{give\text{-}father}(t')$ where $t'$ is the unique thread in $C$ such that $\mathit{give\text{-}id}(t') = \mathit{give\text{-}father}(t)$, otherwise.

Given a configuration $C$, we say a computation unit $(l,\mathit{id})^{(l',\mathit{id}')}$ is alive in $C$ if and only if, there exists an ambient thread $t$ in $C$ such that: $\mathit{give\text{-}id} = (l,\mathit{id})$, $\mathit{give\text{-}father} = (l',\mathit{id}')$. The set of the computation units that are alive in $C$ is written $\mathit{alive}(C)$.

We define an abstraction $\Pi_{\mathscr{P}_0}$ which maps each concrete configuration $C \in \mathscr{C}$ to a function $f$ mapping each alive computation unit $\mathit{loc}^{\mathit{loc}'}$ to the set $\wp(C)$ of threads both in $C$ if and in the computation unit $\mathit{loc}^{\mathit{loc}'}$. The function $\Pi_{\mathscr{P}_0}(C)$ is defined as follows:

$$\left[\mathit{loc}^{\mathit{loc}'} \in \mathit{alive}(C) \mapsto \{t \in C \mid \mathit{give\text{-}father}(t) = \mathit{loc},\ \mathit{give\text{-}big\text{-}father}_C(t) = \mathit{loc}'\}\right].$$

It worth noting that $\Pi_{\mathscr{P}_0}(C)$ defines a partition of the threads in $C$ according to their location.

We introduce the abstract domain $\mathscr{P}_0$ mapping each computation unit to a set of threads. The abstract domain $\mathscr{P}_0$ is related to the concrete domain $\wp(\Sigma^* \times \mathscr{C})$ via a concretization function $\gamma_{\mathscr{P}_0}$. For any function $f \in \mathscr{P}_0$, the concretization $\gamma_{\mathscr{P}_0}(f)$ is defined as:

$$\left\{(u,C) \in \Sigma^* \times \mathscr{C} \;\middle|\; \Pi_{\mathscr{P}_0}(C) \in f\right\}.$$

The abstract domain $\mathscr{P}_0$ may use an unbounded number of classes. To get a decidable abstract semantics, we merge the description of all computation units that have the same labels. An abstract computation unit is given by two labels $l \in \mathscr{L}_{amb}$ and $l' \in \mathscr{L}_{amb}$, and is written $l^{l'}$. We denote by UNIT the set of abstract computation unit. The abstraction function $\Pi_{unit}$ maps each computation unit $(l,\mathit{id})^{(l',\mathit{id}')}$ to the abstract computation unit $l^{l'}$. We introduce the abstract domain $\mathscr{P}_1$ mapping each abstract computation unit to a set of threads. The abstraction function $\Pi_{\mathscr{P}_0}$ maps each function $f \in \mathscr{P}_0$ to the element $\Pi_{\mathscr{P}_0}(f) \in \mathscr{P}_1$ that is defined as follows:

$$\Pi_{\mathscr{P}_1}(f)(l^{l'}) = \bigcup \left\{ f(\mathit{loc}^{\mathit{loc}'}) \;\middle|\; \begin{array}{l} \mathit{loc}^{\mathit{loc}'} \in \mathit{Dom}(f) \\ \Pi_{unit}(\mathit{loc}^{\mathit{loc}'}) = l^{l'} \end{array} \right\}.$$

The abstract domain $\mathscr{P}_1$ is related to the abstract domain $\wp(\mathscr{P}_0)$ via a concretization function $\gamma_{\mathscr{P}_1}$. For any function $g \in \mathscr{P}_1$, the concretization $\gamma_{\mathscr{P}_1}(g)$ is defined as $\{f \in \mathscr{P}_0 \mid \Pi_{\mathscr{P}_1}(f) \in g\}$.

#### 10.2.1.3 Numerical domain

Now, we use a parametric numerical domain to abstract the set of threads that is associated to each abstract computation unit. We introduce a set $\Sigma_{sub}$ of extra variables. These variables are related to transitions labels. But unlike in the occurrence analysis, we do not associate them with a meaning. We relate the set[1] $\Sigma$ to a smaller set $\Sigma_{sub}$ by an onto map $\psi$. Let $\mathscr{V}_c$ be the set $\mathscr{L}_{\mathrm{p}} \cup \mathscr{L}_{amb} \cup \Sigma_{sub}$.

Then, we consider $\wp(\mathbb{N}^{\mathscr{V}_c})$, the complete lattice of sets of function from the set $\mathscr{V}_c$ into the set of the natural numbers. The domain $\wp(\mathbb{N}^{\mathscr{V}_c})$ is used to describe the set of threads in each class of the partition. Thus, we introduce the abstract domain $\mathscr{P}_2$ mapping each abstract computation unit to an element $A \in \wp(\mathbb{N}^{\mathscr{V}_c})$. The abstract domain $\mathscr{P}_2$ is related to the abstract domain $\wp(\mathscr{P}_1)$ via a concretization function $\gamma_{\mathscr{P}_2}$. For any function $h \in \mathscr{P}_2$, the concretization[2] $\gamma_{\mathscr{P}_2}(h)$ is defined as:

$$\left\{ g \in \mathscr{P}_1 \;\middle|\; \begin{array}{l} \forall l, l' \in \mathscr{L}_{amb}, \forall U \in g(l^{l'}),\ \exists f \in h(l^{l'}),\ \text{such that:} \\ \forall p \in \mathscr{L}_{\mathrm{p}} \cup \mathscr{L}_{amb},\ Card\{t \in U \mid \textit{give-pp}(t) = p\} = f_p \end{array} \right\}.$$

Then, we introduce a generic pre-order $(\mathscr{N}_{\mathscr{V}_c}, \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}})$ to represent sets of natural number families indexed by $\mathscr{V}_c$. It is related to $\wp(\mathbb{N}^{\mathscr{V}_c})$ via a monotonic concretization $\gamma_{\mathscr{N}_{\mathscr{V}_c}}$. Furthermore, we introduce several generic primitives: a representation $\bot_{\mathscr{N}_{\mathscr{V}_c}}$ of the empty set, an abstract union $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}$, a widening operator $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$, an abstract counterpart $+^\sharp$ to the binary addition, an abstract counterpart $-^\sharp$ to the subtraction, an abstract synchronization $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}$. We also require the abstraction of some elementary families: an abstract element $0_{\mathscr{N}_{\mathscr{V}_c}}$ to represent the singleton containing the 0 family which associates 0 to each element in $\mathscr{V}_c$, and $\forall v \in \mathscr{V}_c$, an abstract element $1_{\mathscr{N}_{\mathscr{V}_c}}(v)$ to represent the singleton containing the family $\delta^v$ which associates 1 to the element $v$ and 0 to any other elements.

These primitives should satisfy the following properties:

1. $\gamma_{\mathscr{N}_{\mathscr{V}_c}}(\bot_{\mathscr{N}_{\mathscr{V}_c}}) = \emptyset$,
2. $\forall a \in \mathscr{N}_{\mathscr{V}_c},\ \bot_{\mathscr{N}_{\mathscr{V}_c}} \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}} a$,
3. $\forall A \in \wp_{\text{finite}}(\mathscr{N}_{\mathscr{V}_c}),\ \sqcup_{\mathscr{N}_{\mathscr{V}_c}}(A) \in \mathscr{N}_{\mathscr{V}_c}$ and $\forall a \in A,\ a \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}} \sqcup_{\mathscr{N}_{\mathscr{V}_c}}(A)$,
4. $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$ satisfies Def. 7.2.1.(7) on page 158,
5. $\forall a, b \in \mathscr{N}_{\mathscr{V}_c},\ a +^\sharp b \in \mathscr{N}_{\mathscr{V}_c}$ and
   $\{(t_v + u_v)_{v \in \mathscr{V}_c} \mid t \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a), u \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(b)\} \subseteq \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a +^\sharp b)$,

[1]We recall that $\Sigma$ is the set of transition labels.

[2]We have no abstraction function, because we have not given any meaning to the variables in $\Sigma_{sub}$.

6. $\forall a, b \in \mathscr{N}_{\mathscr{V}_c}$, $(a -^\sharp b) \in \mathscr{N}_{\mathscr{V}_c}$ and
$$\left\{ (t_v - u_v)_{v \in \mathscr{V}} \;\middle|\; \begin{array}{l} t \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a),\, u \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(b), \\ \forall v \in \mathscr{V}_c,\, t_v \geqslant u_v \end{array} \right\} \subseteq \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a -^\sharp b),$$

7. $\forall a \in \mathscr{N}_{\mathscr{V}_c}$, $t \in \mathbb{N}^{\mathscr{V}_c}$, $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}(t, a) \in \mathscr{N}_{\mathscr{V}_c}$ and
$\{u \mid u \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}(a), u_v \geq t_v,\, \forall v \in \mathscr{V}_c\} \subseteq \gamma_{\mathscr{N}_{\mathscr{V}_c}}(\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}(t, a))$,

8. $0_{\mathscr{N}_{\mathscr{V}_c}} \in \mathscr{N}_{\mathscr{V}_c}$ and $(0)_{i \in \mathscr{V}_c} \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}\left(0_{\mathscr{N}_{\mathscr{V}_c}}\right)$,

9. $\forall v \in \mathscr{V}_c$, $1_{\mathscr{N}_{\mathscr{V}_c}}(v) \in \mathscr{N}_{\mathscr{V}_c}$ and $(\delta_i^v)_{i \in \mathscr{V}_c} \in \gamma_{\mathscr{N}_{\mathscr{V}_c}}\left(1_{\mathscr{N}_{\mathscr{V}_c}}(v)\right)$,
where $\delta_i^v = \begin{cases} 1 & \text{if } i = v \\ 0 & \text{otherwise.} \end{cases}$

Roughly speaking, the operator $+^\sharp$ is an abstract counterpart to the component-wise addition, the operator $-^\sharp$ is an abstract counterpart to the component-wise subtraction, and the primitive $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}$ is used to extract from an abstract value the representation of the configurations which simultaneously contain all the threads required by a computation step.

#### 10.2.1.4 Abstract domain

We introduce the abstract domain $\mathscr{P} = \textsc{unit} \to \mathscr{N}_{\mathscr{V}_c}$. An abstract element $f \in \mathscr{P}$ relates each abstract computation unit $l^{l'} \in \textsc{unit}$ to an abstraction of the content of any concrete computation unit $(l, id)^{(l', id')}$. Each abstract element $f \in \mathscr{P}$ is related with the set $\gamma_{\mathscr{P}}(f)$ that is defined as follows:

$$C \in (\gamma_{\mathscr{P}_0} \circ \gamma_{\mathscr{P}_1})\left(\left[cu \in \textsc{unit} \mapsto \gamma_{\mathscr{N}_{\mathscr{V}_c}}(f(cu))\right]\right).$$

*Remark* 10.2.1. For any $f \in \mathscr{P}$ and any configuration $(u, C) \in \gamma_{\mathscr{P}}(f)$. A computation unit *cu* may be alive in $C$ only if $f(\Pi_{unit}(cu)) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$.

Partial order $\sqsubseteq_{\mathscr{P}}$ (resp. abstract union $\sqcup^{\mathscr{P}}$, bottom elements $\bot_{\mathscr{P}}$, and widening operator $\nabla_{\mathscr{P}}$) applies the partial order $\sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}$ (resp. the abstract union $\sqcup_{\mathscr{N}_{\mathscr{V}_c}}$, the bottom element $\bot_{\mathscr{N}_{\mathscr{V}_c}}$, and the widening operator $\nabla_{\mathscr{N}_{\mathscr{V}_c}}$) component-wise.

### 10.2.2 Abstract transition system

Now, we define the abstract transition system. The initial configuration is defined in Fig. 10.3 and abstract transition rules are given in Figs. 10.5, 10.6, 10.7 and

$$C_0^{\mathscr{P}} = \left(\beta_2^{\mathscr{P}}(\mathscr{S}, \mathrm{top})\right)\left[\mathrm{top}^{\mathrm{top}} \mapsto \beta_1^{\mathscr{P}}(\mathscr{S})\right].$$

Figure 10.3: Initial abstract configuration.

$$\begin{array}{rcl}
\beta_1^{\mathscr{P}}(n^i[P]) & = & 1_{\mathscr{N}_{\mathscr{V}_c}}(i) \\
\beta_1^{\mathscr{P}}(P \mid Q) & = & \beta_1^{\mathscr{P}}(P) +^{\sharp} \beta_1^{\mathscr{P}}(Q) \\
\beta_1^{\mathscr{P}}((\nu\, n)P) & = & \beta_1^{\mathscr{P}}(P) \\
\beta_1^{\mathscr{P}}(M) & = & 1_{\mathscr{N}_{\mathscr{V}_c}}(lab(M)) \\
\beta_1^{\mathscr{P}}(io) & = & 1_{\mathscr{N}_{\mathscr{V}_c}}(lab(io)) \\
\beta_1^{\mathscr{P}}(\mathbf{0}) & = & 0_{\mathscr{N}_{\mathscr{V}_c}} \\
\\
\beta_2^{\mathscr{P}}(n^i[P], s) & = & \sqcup^{\mathscr{P}}\{\beta_2^{\mathscr{P}}(P, i); [(i,s) \mapsto \beta_1^{\mathscr{P}}(P)]\} \\
\beta_2^{\mathscr{P}}(P \mid Q, s) & = & \sqcup^{\mathscr{P}}\{\beta_2^{\mathscr{P}}(P, s); \beta_2^{\mathscr{P}}(Q, s)\} \\
\beta_2^{\mathscr{P}}((\nu\, n)P, s) & = & \beta_2^{\mathscr{P}}(P, s) \\
\beta_2^{\mathscr{P}}(M, s) & = & [\_ \mapsto \bot_{\mathscr{N}_{\mathscr{V}_c}}] \\
\beta_2^{\mathscr{P}}(io, s) & = & [\_ \mapsto \bot_{\mathscr{N}_{\mathscr{V}_c}}] \\
\beta_2^{\mathscr{P}}(\mathbf{0}, s) & = & [\_ \mapsto \bot_{\mathscr{N}_{\mathscr{V}_c}}]
\end{array}$$

Figure 10.4: Abstract continuations.

10.8. These definitions use auxiliary primitives $\beta_1^{\mathscr{P}}$ and $\beta_2^{\mathscr{P}}$ which abstracts continuation launching. These primitives are defined in Fig. 10.4. The primitive $\beta_1^{\mathscr{P}}$ gives an abstraction of the threads that are launched in a current location. The primitive $\beta_2^{\mathscr{P}}$ gives an abstraction of the threads that are launched in new created ambients.

Abstract *in* migration is described in Fig. 10.5. It requires three threads: the migrating ambient is labeled $i$, the target ambient is labeled $j$, the capability is labeled $k$. The migration is enabled whenever there exist two instances of ambients labeled $s$ and $s'$ such that the following spatial conditions are satisfied:

- the ambient $s$ is in the ambient $s'$ and contains both the ambient $i$ and the ambient $j$;
- the ambient $i$ is in the ambient $s$ and contains the thread $k$;
- the content of the ambient $j$ when surrounded by the ambient $s$ is defined.

In such a case, the abstract computation consists in moving the ambient $i$ inside the ambient $j$:

- an ambient $i$ inside an ambient $j$ may now contain any previous potential content of an instance of ambient $i$ located in an instance of ambient $s$ where

Let $f \in \mathscr{P}$ be an abstract element.
Let $i \in \mathscr{L}_{amb}$, $j \in \mathscr{L}_{amb}$, and $k \in \mathscr{L}_{\mathrm{p}}$ be three labels.
We denote by *context* the set of pairs $(s,s') \in \mathscr{L}_{amb}^2$ such that:

- $\mathrm{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^k], f(i^s)) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$;
- $f(j^s) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$;
- $\mathrm{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_i^x + \delta_j^x], f(s^{s'})) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$.

If $act(i) = act(j) = ambient$, $act(k) = in$, and $context \neq \emptyset$,
then:

$$f \overset{in(i,j,k)}{\rightsquigarrow}_{\mathscr{P}} \sqcup^{\mathscr{P}} \left\{ f_s^{(a,b)} \;\middle|\; \begin{array}{l} \exists s',\ (s,s') \in context, \\ (a,b) = (i,j) \text{ or } (a,b) = (j,s) \end{array} \right\} \cup \{ f; \beta_2^{\mathscr{P}}(cont(k), i) \}$$

where:

- $f_s^{(i,j)} = [i^j \mapsto A(i,j,s) +^\sharp T(i,j,s)]$,
  where $\begin{cases} A(i,j,s) = \mathrm{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^k], f(i^s)), \\ T(i,j,s) = \beta_1^{\mathscr{P}}(cont(k)) -^\sharp 1_{\mathscr{N}_{\mathscr{V}_c}}(k) +^\sharp \psi(in(i,j,k)), \end{cases}$
- $f_s^{(j,s)} = [j^s \mapsto f(j^s) +^\sharp 1_{\mathscr{N}_{\mathscr{V}_c}}(i) +^\sharp \psi(in(i,j,k))]$;

Figure 10.5: Abstract in migration.

an occurrence of thread at program point $k$ has been replaced with the thread continuation: this continuation is given by the function $\beta_1^{\mathscr{P}}$; we also increment the variable $\psi(in(i,j,k))$ in this abstract content; this incrementation has no meaning but helps in expressing more invariants;

- an ambient $j$ inside an ambient $s$ may now contain any previous potential content of an instance of ambient $j$ inside an instance of ambient $s$ where an instance of thread at program point $i$ has been created (we also increment the variable $\psi(in(i,j,k))$ in this abstract content);
- new created ambients are created; their initial content is statically known: they are given by the function $\beta_2^{\mathscr{P}}$.

Abstract *out* migration is described in Fig. 10.6. It requires three threads: the migrating ambient is labeled $i$, the expelling ambient is labeled $j$, the capability is labeled $k$. The migration is enabled whenever there exist two ambients labeled $s$ and $s'$ such that the following spatial conditions are satisfied:

Let $f \in \mathscr{P}$ be an abstract element.
Let $i \in \mathscr{L}_{amb}$, $j \in \mathscr{L}_{amb}$, and $k \in \mathscr{L}_{\mathrm{p}}$ be three labels.
We denote by *context* the set of pairs $(s,s') \in \mathscr{L}_{amb}^2$ such that:

1. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^j], f(s^{s'})) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$;
2. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^i], f(j^s)) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$;
3. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^k], f(i^j)) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$.

If $act(i) = act(j) = ambient$, $act(k) = out$, and $context \neq \emptyset$,
then:

$$f \overset{out(i,j,k)}{\rightsquigarrow}_{\mathscr{P}} \sqcup^{\mathscr{P}} \left\{ f_{(s,s')}^{(a,b)} \;\middle|\; \begin{array}{l} (s,s') \in context \\ (a,b) = (i,s) \text{ or } (a,b) = (s,s') \\ \text{or } (a,b) = (j,s) \end{array} \right\} \cup \{ f; \beta_2^{\mathscr{P}}(cont(k), i)\}$$

where:

- $f_{(s,s')}^{(i,s)} = [i^s \mapsto A(i,j,s) +^{\sharp} T(i,j,s)]$

  where $\begin{cases} A(i,j,s) = \textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^k], f(i^j)) \\ T(i,j,s) = \beta_1^{\mathscr{P}}(cont(k))) -^{\sharp} 1_{\mathscr{N}_{\mathscr{V}_c}}(k) +^{\sharp} \psi(out(i,j,k)), \end{cases}$

- $f_{(s,s')}^{(j,s)} = [j^s \mapsto B(i,j,s) +^{\sharp} U(i,j,s)]$

  where $\begin{cases} B(i,j,s) = \textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^i], f(j^s)) \\ U(i,j,s) = \psi(out(i,j,k)) -^{\sharp} 1_{\mathscr{N}_{\mathscr{V}_c}}(i), \end{cases}$

- $f_{(s,s')}^{(s,s')}(s^{s'}) = C(i,j,s,s')$,

  where $C(i,j,s,s') = \textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^j], f(s^{s'})) +^{\sharp} 1_{\mathscr{N}_{\mathscr{V}_c}}(i) +^{\sharp} \psi(out(i,j,k))$.

Figure 10.6: Abstract out migration.

- the ambient $s$ is in the ambient $s'$ and contains the ambient $j$;
- the ambient $j$ is in the ambient $s$ and contains the ambient $i$;
- the ambient $i$ is in the ambient $j$ and contains the thread $k$.

In such a case, the abstract computation consists in moving the ambient $i$ inside the ambient $s$:

- an ambient $i$ inside an ambient $s$ may now contain any previous potential content of an instance of ambient $i$ located in an instance of ambient $j$ where an occurrence of thread at program point $k$ has been replaced with the thread continuation: this continuation is given by the function $\beta_1^{\mathscr{P}}$; we also increment the variable $\psi(out(i,j,k))$ in this abstract content; this incrementation has no meaning but helps in expressing more invariants;
- an ambient $j$ inside an ambient $s$ may now contain any previous potential content of an instance of ambient $j$ inside an instance of ambient $s$ where an instance of ambient $i$ has been removed (we also increment the variable $\psi(out(i,j,k))$ in this abstract content);
- an ambient $s$ inside an ambient $s'$ may now contain any previous potential content of an instance of ambient $s$ inside an instance of ambient $s'$ where an instance of ambient $i$ has been added (we also increment the variable $\psi(out(i,j,k))$ in this abstract content);
- new created ambients are created; their initial content is statically known: they are given by the function $\beta_2^{\mathscr{P}}$.

Abstract dissolution is described in Fig. 10.7. It requires two threads: the capability is labeled $i$, the ambient is labeled $j$. The dissolution is enabled whenever there exist two ambients labeled $s$ and $s'$ such that the following spatial conditions are satisfied:

- the ambient $s$ is in the ambient $s'$ and contains both the thread $i$ and and the ambient $j$;
- the content of the ambient $j$ when surrounded by the ambient $s$ is defined.

In such a case, the abstract computation consists in updating the content of $s$:

- an ambient $s$ inside an ambient $s'$ may now contain any previous potential content of an instance of ambient $s$ located in an instance of ambient $s'$ where we have added any potential content of an instance of an ambient $j$ inside located in an instance of an ambient $s$ and the thread continuation;

Let $f \in \mathscr{P}$ be an abstract element.
Let $i \in \mathscr{L}_{\text{p}}$ and $j \in \mathscr{L}_{amb}$ be two labels.
We denote by *context* the set of pairs $(s,s') \in \mathscr{L}^2_{amb}$ such that:

1. $\text{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta^i_x + \delta^j_x], f(s^{s'})) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$;
2. $f(j^s) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$.

If $act(i) \in \{open; !open\}$, $act(j) = ambient$, and $context \neq \emptyset$, then:

$$f \overset{act(i)(i,j)}{\rightsquigarrow}_{\mathscr{P}} \sqcup^{\mathscr{P}} \bigcup \left\{ \left\{ f^{(s,s')}_{(s,s')}; \beta^{\mathscr{P}}_2(cont(i),s); f^s_{(s,s')} \right\} \mid (s,s') \in context \right\} \cup f$$

where:

- $f^{(s,s')}_{(s,s')} = [s^{s'} \to \sqcup_{\mathscr{N}_{\mathscr{V}_c}} \{ f(s^{s'}); A(i,j,s,s') +^{\sharp} T(i,j,s,s') +^{\sharp} \psi(act(i)(i,j)) \}]$,

  where $\begin{cases} A(i,j,s,s') = \text{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta^i_x + \delta^j_x], f(s^{s'})), \\ T(i,j,s,s') = f(j^s) -^{\sharp} r -^{\sharp} 1_{\mathscr{N}_{\mathscr{V}_c}}(j) +^{\sharp} \beta^{\mathscr{P}}_1(cont(i)), \\ r = \begin{cases} 0_{\mathscr{N}_{\mathscr{V}_c}}, & \text{if } act(i) = !open \\ 1_{\mathscr{N}_{\mathscr{V}_c}}(i), & \text{otherwise,} \end{cases} \end{cases}$

- $f^s_{(s,s')} = [x^s \mapsto \sqcup_{\mathscr{N}_{\mathscr{V}_c}} \{ f(x^s), f(x^j) \}$, for any $x \in \mathscr{L}_{amb}]$.

Figure 10.7: Abstract dissolution.

this continuation is given by the function $\beta^{\mathscr{P}}_1$; we also increment the variable $\psi(open(i,j))$ in this abstract content; this incrementation has no meaning but helps in expressing more invariants; moreover, in case the capability thread is not a resource, we decrement the occurrence number of threads at program point $i$; in the case when the capability is a resource, it is still available after the computation step;

- an ambient $x$ inside an ambient $s$ may contain any previous potential content of an ambient $x$ inside an ambient $s$, but also any previous potential content of an ambient $x$ inside an ambient $j$. These contents may not be merged because due to marker unambiguity, two threads in a concrete configuration in $\mathscr{C}$ that have the same location are also in the same computation unit;

- new created ambients are created; their initial content is statically known: they are given by the function $\beta^{\mathscr{P}}_2$.

Let $f \in \mathscr{P}$ be an abstract element.
Let $i$, $j$ be two labels in $\mathscr{L}_{\mathrm{p}}$.
We denote by *context* the set of pairs $(s,s') \in \mathscr{L}_{amb}^2$ such that:

$$\mathrm{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^i + \delta_x^j], f(s^{s'})) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}.$$

If $act(i) \in \{input; !input\}$, $act(j) = output$, and $context \neq \emptyset$, then:

$$f \overset{act(i)(i,j)}{\rightsquigarrow}_{\mathscr{P}} \sqcup^{\mathscr{P}} \{f'_{(s,s')} \cup \beta_2^{\mathscr{P}}(cont(i),s) \mid (s,s') \in context\};$$

where:

- $f'_{(s,s')}(s^{s'}) = \sqcup_{\mathscr{N}_{\mathscr{V}_c}} \left\{ f(s^{s'}); A(i,j,s,s') +^\sharp T(i,j,s,s') \right\}$,

  where:

$$\begin{cases} A(i,j,s,s') = \mathrm{SYNC}_{\mathscr{N}_{\mathscr{V}_c}}([x \mapsto \delta_x^i + \delta_x^j], f(s^{s'})), \\ T(i,j,s,s') = \beta_1^{\mathscr{P}}(cont(i)) -^\sharp r -^\sharp 1_{\mathscr{N}_{\mathscr{V}_c}}(j) +^\sharp \psi(i,j), \\ r = \begin{cases} 0_{\mathscr{N}_{\mathscr{V}_c}} & \text{if } act(i) = !input, \\ 1_{\mathscr{N}_{\mathscr{V}_c}}(i) & \text{otherwise}, \end{cases} \end{cases}.$$

- $f'_{(s,s')}(x) = f(x)$, otherwise.

Figure 10.8: Abstract communication.

Abstract communication is described in Fig. 10.8. It requires two threads: the input thread is labeled $i$, the output thread is labeled $j$. The communication is enabled whenever there exist two ambients labeled $s$ and $s'$ such that the following spatial conditions are satisfied:

- the ambient $s$ is in the ambient $s'$ and contains both the thread $i$ and the thread $j$;

In such a case, the abstract computation consists in updating the content of $s$: we take into account threads that are consumed, and launch the abstract continuation (this continuation is given by the function $\beta_1^{\mathscr{P}}$); new created ambients are created; their initial content is statically known: they are given by the function $\beta_2^{\mathscr{P}}$.

**Theorem 10.2.2.** *$(\mathscr{P}, \sqsubseteq_{\mathscr{P}}, \sqcup^{\mathscr{P}}, \bot_{\mathscr{P}}, \gamma_{\mathscr{P}}, C_0^{\mathscr{P}}, \rightsquigarrow_{\mathscr{P}}, \nabla_{\mathscr{P}})$ is an abstraction.*

### 10.2.3 Prototype and analysis example

#### 10.2.3.1 Prototype

We now present an example of analysis result. This result has been automatically computed by using the *amb*-s.a. prototype [33] for systems that are written in the *ambient* calculus. The analyzer proposes the choice between several numerical domains: the intervals [22], the product between the intervals and the affine equalities [47], the octagons [55, 58], and the abstract multi-sets [43] (based on the use of disjunctive completion).

#### 10.2.3.2 Example

We now describe results obtained on an example. For the sake of brevity and simplicity, we do not present linear constraints. They are not of interest when considering the result as they are used internally to refine the interval information.

**Example 10.2.3.** *We analyze the system that is given in Ex. 9.4.4 on page 260. We label this system in Fig. 9.4.4.*

*The analyzer detects that a* **token** *ambient (created at program point* **22***) contains:*

- *exactly one thread at program point* **24** *(i.e. the capability to enter inside the* **server** *ambient), when the token ambient is at top level;*
- *no thread, when the token ambient is inside the* **server** *ambient;*
- *exactly one thread at program point* **24** *(i.e. the capability to enter inside the* **server** *ambient), when the token ambient is inside the* **client** *ambient;*
- *exactly one thread at program point* **23** *(i.e. the capability to exit a packet), when the token ambient is inside a packet (i.e. an ambient created at program point* **32***) ambient.*

*This is a very accurate description of the content of the token, during the computation of the server instance. Since there is at most one thread in each token and since we have capture the fact that there is at most three tokens in the system (Cf. Ex. 9.4.4 on page 260), a reduced product between occurrence counting analysis and thread partitioning analysis would prove that there is at most three simultaneous instances of threads at program point* **23** *or at program point* **24***.*

## 10.3 Generalization

We extend this analysis to deal with the models that are encoded in our meta-language.

```
$(\nu\ \text{request})(\nu\ \text{make})(\nu\ \text{server})(\nu\ \text{duplicate})(\nu\ \text{instance})$
$(\nu\ \text{answer})(\nu\ \text{client})(\nu\ \text{token})$
$($
 $\text{server}^{11}[$
           $!\mathit{open}^{12}\text{duplicate}.0$
          $|$
           $!(k)^{13}.\mathit{open}^{14}\text{token}.$
               $\text{instance}^{15}[$
                            $\mathit{in}^{16}k.\mathit{open}^{17}\text{request}.(\mathit{rep})^{18}.$
                               $($
                                $\text{answer}^{19}[<\mathit{rep}>^{20}]$
                               $|$
                                $\mathit{out}^{21}\text{server}.\text{token}^{22}[\mathit{out}^{23}k.\mathit{in}^{24}\text{server}.0]$
                               $)$
                           $]$
          $|$
           $\text{token}^{25}[0]\ |\ \text{token}^{26}[0]\ |\ \text{token}^{27}[0]$
          $]$
$|$
 $\text{client}^{28}[!(x)^{29}.((\nu\ q)(\nu\ p)$
                   $p^{32}[$
                        $\mathit{out}^{33}\text{client}.0$
                       $|$
                        $\text{request}^{34}[<q>^{35}\ [<q>]$
                       $|$
                        $\mathit{open}^{36}\text{instance}.0$
                       $|$
                        $\mathit{in}^{37}\text{server}.\text{duplicate}^{38}[\mathit{out}^{39}p.<p>^{40}]$
                       $]$
                  $|$
                   $<\text{make}>^{41}$
                  $)$
          $|$
           $<\text{make}>^{42}$
          $]$
$)$
```

Figure 10.9: An *ftp*-server with tokens in mobile ambients.

## 10.3.1 Concrete partitioning

Let $B$ be a finite set of indice. We define the set *unit* of computation units as the set $B \rightarrow \mathscr{L} \times \mathscr{M}$ mapping each index $b \in B$ to a value $v \in \mathscr{L} \times \mathscr{M}$.

We consider a function *give-index* mapping each program point $p$ to a computation unit $u \in B \rightarrow I(p)$. Then, the computation unit of a thread $t = (p, id, E)$ at program point $p$ is defined as the function $[b \in B \rightarrow E(\textit{give-index}(p)(b))]$, this computation unit is written $\textit{give-unit}(t)$.

*Remark* 10.3.1. We assume that the computation unit of a thread $t$ is fully defined by the program point and the environment of this thread (and not by the other threads in the system configuration). Otherwise, we can enrich thread environments so that they map new variables to the values which are required to define thread computation units. In that case, we also refine formal rules so that these rules take into account the constraints about the values of these new variables. New rules also allows the description of value passing among these extra variables.

**Example 10.3.2 (Computation units in mobile ambients).** *In mobile ambients, the computation unit of a thread $t$ is the pair $(loc, loc')$ (that we denote $loc^{loc'}$) where $loc$ is the location of the thread and $loc'$ is the location of the ambient that contains the thread $t$. The set $B$ is chosen as the set $\{1;2\}$. We enrich thread environments with an extra variable $loc'$ that denotes the location of the surrounding ambient. Then, for any program point $p$, the function give-index$(p)$ is defined as the function $[1 \mapsto loc, 2 \mapsto loc']$. Then, we have to refine formal rules, so that they take into account constraints about the location of the surrounding ambient and so that they describe how the location of the surrounding ambient is updated during computation steps. We may also use more ancestors to define the computation unit. The more ancestors we consider, the more precise the analysis is.*

In the set $B$, some indice may be redundant. For instance, in the case of the *ambient* calculus, the computation unit is given by both the surrounding ambient and the location of this surrounding ambient. But, by construction, if two threads are in a same ambient, the location of their surrounding ambient is the same. Having the location of the surrounding ambient in the computation unit allows for a more precise partitioning at the abstract level. We denote by $B_s$ the set of indice $b \in B$ that satisfy: for any configuration $C \in \mathscr{C}$, for any thread $t_1 = (p_1, id_1, E_1)$ and $t_2 = (p_2, id_2, E_2)$ in the configuration $C$, if $E_1(\textit{give-index}(p_1)(b)) = E_2(\textit{give-index}(p_2)(b))$ for any $b \in B_s$, then $\textit{give-unit}(t_1) = \textit{give-unit}(t_2)$.

### 10.3.2 Abstract partitioning

To get a decidable abstraction, we merge the description of the computation units that have the same labels. We define the set UNIT of abstract units as $B \rightarrow \mathscr{L}$. The abstraction function $\Pi_{unit}$ maps each computation unit $[b \in B \mapsto (l_b, id_b)] \in unit$ to the abstract computation unit $[b \mapsto l_b] \in$ UNIT.

### 10.3.3 Environment and counting domains

We want to abstract both dynamic properties about the linkage of threads and concurrency properties. We compose several abstract domains to build our main domain.

1. We assume that we are given a family $(Atom^{\sharp}_{V})_{V \subseteq \mathscr{V}}$ of abstract domains for describing markers and environments of single threads. We also suppose that we are given a family $(Molecule^{\sharp}_{(V_i)})_{(V_i) \in \wp(\mathscr{V})^*}$ of abstract domains for describing markers and environments of thread tuples. These two families are fitted with their corresponding primitives, as defined in Sects. 8.1.1.1 and 8.1.2 on page 169.

   Moreover, we suppose that we are given a primitive *force-lab* to enforce constraints about the labels of the variable values. For any tuple of interface $(V_i) \in \wp(\mathscr{V})^n$, for any abstract molecule $m \in Molecule^{\sharp}_{(V_i)}$, for any set $S = \{((x_k, j_k), l_k) \mid k \in K\}$ of elements $((x, j), l)$ such that $1 \leq j \leq n$, $x \in V_i$ and $l \in \mathscr{L}$, the abstract element $\textit{force-lab}(S, m)$ is an abstract molecule in $Molecule^{\sharp}_{(V_i)}$ that satisfies:

   $$\left\{(id_i, E_i) \in \gamma_{(V_i)}(m) \;\middle|\; \forall k \in K,\, fst(E_{j_k}(x_k)) = l_k\right\} \subseteq \gamma_{(V_i)}(\textit{force-lab}(S, m)).$$

2. We choose a set $\Sigma_{sub}$ of factious counters. We map each transition label $\lambda \in \Sigma$ to a factious counter $\psi(\lambda) \in \Sigma_{sub}$. The function $\psi$ is not necessarily injective. We define a set $\mathscr{V}_c$ of counters as $\mathscr{L}_{\mathrm{p}} \cup \Sigma_{sub}$. Counters $p \in \mathscr{L}_{\mathrm{p}}$ denote occurrence number of thread at program point $p$. Other counters have no meaning. We assume that we are given a numerical domain $\mathscr{N}_{\mathscr{V}_c}$, fitted with its primitives, as defined in Sect. 9.2 on page 249.

### 10.3.4 Main domain

Our main domain is a Cartesian product between a domain that describes the pair marker/environment of each thread in the system and a domain that describes the occurrence number of threads inside non-empty computation units.

Let $\mathscr{A}_{env} = (\mathscr{C}^{\sharp}_{env}, \sqsubseteq_{env}, \sqcup_{env}, \bot_{env}, \gamma_{env}, C0_{env}, \rightarrow_{env}, \nabla_{env})$ be the abstraction defined by the families $(Atom^{\sharp}_{V})$ and $(Molecule^{\sharp}_{(V_i)})$ as in Sect. 8.1 on page 169. Our main domain $\mathscr{C}^{\sharp}_{part}$ is defined as the product $\mathscr{C}^{\sharp}_{env} \times (\text{UNIT} \rightarrow \mathscr{N}_{\mathscr{V}_c})$. The abstract domain $\mathscr{C}^{\sharp}_{part}$ is related to the concrete domain $\wp(\Sigma^* \times \mathscr{C})$ by a concretization function $\gamma_{part}$. For each abstract element $(\text{ENV}, \text{CU}) \in \mathscr{C}^{\sharp}_{part}$, the set $\gamma_{part}(\text{ENV}, \text{CU})$ contains any configuration $(v, C) \in \Sigma^* \times \mathscr{C}$ that satisfies:

1. $(v, C) \in \gamma_{env}(\text{ENV})$;
2. for any computation unit $u \in unit$, there exists a function $t \in \{(0) \in \mathbb{N}^{\mathscr{V}_c}\} \cup (\gamma_{\mathscr{N}_{\mathscr{V}_c}}(\text{CU}(\Pi_{unit}(u))))$ such that the occurrence number of threads at program point $p$ in the computation unit $u$ is equal to $t(p)$, i.e. :

$$t(p) = Card(\{(p, id, E) \in C \mid give\text{-}unit(p, id, E) = u\}).$$

We define the pre-order $\sqsubseteq_{part}$, the abstract union $\sqcup_{part}$, the bottom element $\bot_{part}$, and the widening operator $\nabla_{part}$ component-wise.

## 10.3.5 Partitioning primitives

Let $n$ be a natural number. Let $(p_i)_{1 \le i \le n} \in \mathscr{L}^n_{\text{p}}$ be a tuple of program points. Let $mol \in Molecule^{\sharp}_{(I(p_i))_{1 \le i \le n}}$ be an abstract molecule that describes threads at program points $(p_i)$.

1. we define the primitive *same-unit* that inserts the constraint that two threads belong to the same computation unit. We use the abstract primitive $\text{SYNC}^{\sharp}$ that is defined on page 172 to enforce equality constraints among the values associated to each index $b \in B$ in both threads. Let $i$ and $j$ be two integers between 1 and $n$. We define the primitive *same-unit* as:

$$same\text{-}unit(i, j, (p_k), mol) \stackrel{\Delta}{=} \text{SYNC}^{\sharp}(S, (p_k), mol),$$

   where $S = \{((give\text{-}index(p_i)(b)), i) = ((give\text{-}index(p_j)(b)), j) \mid b \in B\}$.

2. we define the primitive *distinct-unit* that inserts disequality constraints about the computation unit of two threads. If the set $B_s$ contains exactly one index, then we use abstract primitive $\text{SYNC}^{\sharp}$ that is defined on page 172 to enforce that the values associated to this index in both threads are distinct. Otherwise, we do not change the abstract molecule. Let $i$, $j$ be two integers between 1 and $n$. We define the primitive *distinct-unit* as:

- if $Card(B_s) = 1$,

  $$distinct\text{-}unit(i, j, (p_k), mol) \triangleq \text{SYNC}^\sharp(S, (p_k), mol),$$

  where:
  $S = \{((give\text{-}index(p_i)(b)), i) \neq ((give\text{-}index(p_j)(b)), j) \mid b \in B_s\}$;
- otherwise,

  $$distinct\text{-}unit(i, j, (p_k), mol) \triangleq S.$$

3. we define the primitive *set-unit* that inserts constraints about the abstract computation unit of a thread. Let $a \in \text{UNIT}$ be an abstract computation unit. Let $i$ be an integer between 1 and $n$. We define the primitive *set-unit* as:

   $$set\text{-}unit(i, a, (p_k), mol) \triangleq force\text{-}lab(S, mol),$$

   where $S = \{((give\text{-}index(p_i)(b)), i), a(b)) \mid b \in B\}$.

## 10.3.6 Abstract operational semantics

We now use these generic primitives to simulate in the abstract the concrete operational semantics.

### 10.3.6.1 Enabling a partial interaction

Let us consider $C^\sharp = (\text{ENV}, \text{CU}) \in \mathscr{C}^\sharp_{part}$ an abstract configuration. Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule. Let $(p^k)_{1 \leq k \leq n} \in \mathscr{L}^n_{\text{p}}$ be an $n$-tuple of program point labels and $(pi^k)_{1 \leq k \leq n} = (s^k, (parameter^k_l)_l, (bd^k_l)_{k,l}, constraints^k, continuation^k)_{1 \leq k \leq n}$ be an $n$-tuple of partial interactions. For any $k \in [\![1; n]\!]$, we choose $Ct^k \in continuation^k$.

We consider a concrete configuration $C \in \gamma_{part}(\text{ENV}, \text{CU})$. We consider an $n$-tuple of threads $(t^i) = (p^i, id^i, E^i) \in C^n$ (each thread $t^i$ is at program point $p^i$). At the concrete level, the computation step labeled $(\mathscr{R}, (t_k, pi_k))$ is enabled if each thread $t_i$ at program point $p^i$, may compute the partial correct interaction and if the synchronization constraints of the rule $\mathscr{R}$ are satisfied. Moreover, each thread $t_i$ is in a computation unit. We consider the equivalence relation $\sim$ over the interval $[\![1; n]\!]$ that is defined as $[i \sim j \iff give\text{-}unit(t_i) = give\text{-}unit(t_j)]$. We denote by $A$ the function that maps each integer $i \in [\![1; n]\!]$ to the abstract computation unit of the thread $t_i$. We know that each computation unit $unit(t_i)$ contains all the threads $t_j$ such that $i \sim j$.

At the abstract level, the equivalence relation $\sim$ and the function $A$ are not known. We compute the set *context* of all pairs $(\sim, A)$ such that there may exist a configuration $C$ in $\gamma_{part}(\text{ENV},\text{CU})$, such that there exists an $n$-tuple of threads $(t^i) = (p^i, id^i, E^i) \in C^n$ (each thread $t^i$ is at program point $p^i$) such that $C$ may perform the computation step labeled $(\mathscr{R}, (t_k, pi_k))$ and such that the tuple $(t^i)$ satisfies the constraints of $(\sim, A)$. We collect several kind of constraints. In Sect. 10.3.6.1.1, we give the syntactic constraints. In Sect. 10.3.6.1.2, we give constraints about environments. In Sect. 10.3.6.1.3, we give constraints about occurrence numbers of threads.

#### 10.3.6.1.1 Syntactic constraints

$$pi^k \in interaction(p^k).$$

**10.3.6.1.2 Constraints about environments** First, we define the reactive molecule $mol_0^1$ as in Sect. 8.1.3.2 on page 174.

$$mol_0^1 \stackrel{\Delta}{=} \text{SYNC}^\sharp(cons, (p^k), mol),$$

where:

- $mol \stackrel{\Delta}{=} \text{INJ}^\sharp(\text{ENV}(p^1)) \bullet \ldots \bullet \text{INJ}^\sharp(\text{ENV}(p^n))$;
- $cons \stackrel{\Delta}{=} \bigcup(\{R_k \mid 0 \leq k \leq n\})$;
- $R_0 \stackrel{\Delta}{=} \{\sigma(X) = \sigma(Y) \mid (X,Y) \in compatibility\}$,
  with $\sigma : \begin{cases} I^k \mapsto (I,k) \\ X_l^k \mapsto (param_l^k, k); \end{cases}$
- $\forall k \in [\![1;n]\!],\ R_k = \{(x,k) \diamond (y,k) \mid x \diamond y \in constraints^k\}$;

Then, we take into account the constraints about computation unit: for any $i$, $j$ such that $1 \leq j \leq i \leq n$, we define the abstract molecule $mol_j^i$ by induction as:

$$\begin{cases} mol_i^i = set\text{-}unit(i, A(i), mol_{i-1}^i) \\ mol_1^{i+1} = \begin{cases} same\text{-}unit(1, i+1, (p^k), mol_i^i) & \text{if } i+1 \sim 1, \\ distinct\text{-}unit(1, i+1, (p^k), mol_i^i) & \text{otherwise,} \end{cases} \\ mol_{j+1}^i = \begin{cases} same\text{-}unit(j+1, i, (p^k), mol_j^i) & \text{if } j+1 \sim i, \\ distinct\text{-}unit(j+1, i, (p^k), mol_j^i) & \text{otherwise,} \end{cases} \end{cases}$$

Thus, we define the interacting molecule $mol(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1 \leq k \leq n}, \sim, A)$ as:

$$mol(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1 \leq k \leq n}, \sim, A) \stackrel{\Delta}{=} mol_n^n$$

The computation step with the contextual constraints given by $\sim$ and $A$ is enabled only if:

$$mol(C^\sharp,\mathscr{R},(p^k,pi^k)_{1\leq k\leq n},\sim,A)\neq\bot_{(I(p^k))}.$$

#### 10.3.6.1.3 Constraints about computation units

Let $i$ be an integer between 1 and $n$. We denote by $C_i$ the equivalence class $\{j\in[\![1;n]\!]\mid i\sim j\}$. We denote by $(t^i_v)_{v\in\mathcal{V}_c}\in\mathbb{N}^{\mathcal{V}_c}$ the tuple such that $t^i_v$ is the cardinal of the set $\{j\in[\![1;n]\!]\mid v=p^j,\ \textit{give-unit}(t_i)=\textit{give-unit}(t_j)\}$. For each counter $v$, if $v\in\mathscr{L}_{\mathrm{p}}$, the counter $v$ denotes a program point. In such a case, the integer $t^i_v$ denotes the number of threads at program point $v$ that are required in the computation unit of the $i$-th interacting thread. We use the primitive $\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}$ to check that these constraints are satisfiable. This way, the abstract computation with the contextual constraints given by $\sim$ and $A$ is enabled only if:

$$\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}((t^i_v)_{v\in\mathcal{V}_c},\textsc{cu}(A(i)))\neq\bot_{\mathcal{N}_{\mathcal{V}_c}},\ \forall i\in[\![1;n]\!].$$

### 10.3.6.2 Marker computation and value passing

Then, we compute the molecule that describes the pairs marker/environment of the threads after value passing and marker computation. For that purpose, we use the primitive that is given in Def. 8.1.2 on page 175. Let $t$ be a partial interaction name type in $\{\textit{replication};\textit{computation};\textit{migration}\}$. Let $n$ be an integer. Let $(p^k)_{1\leq k\leq n}$ be an $n$-tuple of program point labels. Let $(bd^k_l)_{k,l}$ be an $n$-tuple of sequences of variables ($bd^k_l$ is associated with the $l$-th variable (in the $k$-th thread) that is bound by the interaction). Let $(param^k_l)_{k,l}$ be an $n$-tuple of sequences of parameters ($param^k_l$ is associated with the $l$-th parameter of the $k$-th thread). Let *v-passing* be a partial map from $\mathcal{V}^Y_f$ into $\mathcal{V}^X_f\cup\mathcal{V}^I_f$. Let $molecule^\sharp\in Molecule^\sharp_{(I(p^i))_{1\leq i\leq n}}$ be the abstraction of $n$ interacting threads, just before computing value passing and marker computation. The abstract molecule after value passing and marker computation is defined as:

$$\textit{marker-value}(t,(p^k)_k,molecule^\sharp,(bd^k_l)_{k,l},(param^k_l)_{k,l},\textit{v-passing}).$$

### 10.3.6.3 Launching continuations

We now describe how we simulate continuation launching at the abstract level. Let $C^\sharp=(\textsc{env},\textsc{cu})\in\mathscr{C}^\sharp_{part}$ be an abstract configuration. Let $\mathscr{R}=(n,\textit{components},\textit{compatibility},\textit{v-passing},\textit{broadcast})$ be a reduction rule. Let $(p^k)_{1\leq k\leq n}\in\mathscr{L}^n_{\mathrm{p}}$ be an $n$-tuple of program point labels and $(pi^k)_{1\leq k\leq n}=$

$(s^k,(parameter_l^k)_l,(bd_l^k)_{k,l},constraints^k,continuation^k)_{1\leq k\leq n}$ be an $n$-tuple of partial interactions. Let $(V_i)_{1\leq k\leq n}$ be an $n$-tuple of interfaces (after value passing). Let $(ct^k)\in\wp(\mathcal{L}_{\mathrm{p}}\times(\mathcal{V}\rightharpoonup\mathcal{L}))$ be an $n$-tuple of continuations, such that $ct^k\in continuation^k$, for any $k$ such that $1\leq k\leq n$. Let $(bd_l^k)_{k,l}$ be an $n$-tuple of sequences of variables ($bd_l^k$ is associated with the $l$-th variable (in the $k$-th thread) that is bound by the interaction). Let $(param_l^k)_{k,l}$ be an $n$-tuple of sequences of parameters ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread). Let $v$-$passing$ be a partial map from $\mathcal{V}_f^Y$ into $\mathcal{V}_f^X\cup\mathcal{V}_f^I$. Let $mol^0\in Molecule^\sharp_{(I(p^i))_{1\leq i\leq n}}$ be an abstract molecule. Let $mol'\in Molecule^\sharp_{(V_i)_{1\leq i\leq n}}$ be another abstract molecule. The abstract molecule $mol^0$ denotes the $n$ interacting threads just before computing value passing and marker computation; the abstract molecule $mol'$ denotes the $n$ interacting threads just after computing value passing and marker computation. We introduce, for any $i$ such that $1\leq i\leq n$, the integer $n_i$ as the cardinal of the set $ct^i$. We also write $ct^i=\{(p^{(i,j)},E_s^{(i,j)})\mid 1\leq j\leq n_i\}$. We define an integer $m$ as the sum $\Sigma_{1\leq i\leq n}n_i$. We introduce $\phi$ as the unique monotonic bijection from the set $\{(i,j)\mid 1\leq i\leq n,\ 1\leq j\leq n_i\}$ (we use the lexical order) into the interval $[\![1;m]\!]$.

#### 10.3.6.3.1 Balance molecule

Unlike in the environment analysis, we do not merge all potential continuations in the abstract level: we do a case analysis according to which continuations are launched. Then, for each choice of potential continuations, we build a molecule that gathers the descriptions of the threads that interact and of the threads that are launched. We call this molecule the balance molecule.

**Definition 10.3.3 (balance molecule).** We gather the description of the interacting threads (before value passing and marker computation) and the description of threads in their continuation as follows. For any $k\in[\![1;m]\!]$, we define the molecule $mol^k$ by induction as:

$$mol^k=mol^{k-1}\bullet\mathrm{GC}^\sharp(I(p^{(i,j)}),update^\sharp(E_s^{(i,j)},\mathrm{PROJ}^\sharp(i,mol'))),$$

where $\phi(i,j)=k$.

Then, we collect constraints about the computation units of interacting threads and about the computation units of launched threads. For each launched thread, each index in the computation unit is associated to a value. Either this value is fresh, or it comes from another interacting thread. In the last case, we collect some constraints: there are two sub-cases, either the value has been passed from an interacting thread to its continuation, or its has been passed by another thread through a communication. We define the balance molecule *balance* as the molecule $\mathrm{SYNC}^\sharp(cons,(q^k)_{1\leq k\leq m+n},mol^m)$, where:

1. the set of constraints *cons* is defined as:

$$\left\{ (x, \phi(i,j)+n) = \sigma(x,i,j) \;\middle|\; \begin{array}{l} 1 \le i \le n,\ 1 \le j \le n_j, \\ \exists b \in B,\ x = \textit{give-index}(p^{(i,j)})(b), \\ x \notin \textit{Dom}(E_s^{(i,j)}) \end{array} \right\},$$

with $\sigma(x,i,j) = \begin{cases} (\textit{param}_b^a, a) & \text{if } \exists j', \begin{cases} bd^i_{j'} = x, \\ \textit{v-passing}(Y^i_{j'}) = X^a_b, \end{cases} \\ (x,i) & \text{otherwise.} \end{cases}$

2. $q^k = \begin{cases} p^k & \text{if } 1 \le k \le n, \\ p^{(i,j)} & \text{if } k > n \text{ and } k - n = \phi(i,j). \end{cases}$

**10.3.6.3.2 Potential computation units and potential labeling** Then, we consider any case of equality relations among the computation units of the threads that are described in the balance molecule, of abstract computation units, and of the formal variable value labeling. For any $k$ such that $1 \le k \le n$, we denote $(a_k, b_k) = \textit{arity}(\textit{components}(k))$. We introduce the set $\mathscr{V}^X_{\mathscr{R}}$ of formal variables as $\{X^k_l \mid 1 \le k \le n,\ 1 \le l \le a_k\}$. Each case is described by an equivalence relation $\sim$ over the interval $[\![1; m+n]\!]$, by a function $A \in [\![1; m+n]\!] \to \textsc{unit}$, and by a function $\textit{lab} \in \mathscr{V}^X_{\mathscr{R}} \to \mathscr{L}$. The equivalence relation $\sim$ means that the $i$-th thread and the $j$-th thread in the balance molecule are in the same computation unit if and only if $i \sim j$. The function $A$ means that the abstract computation unit of the $i$-th thread is $A(i)$. The function *lab* means that each formal variable $X$ is associated with a value the label of which is $\textit{lab}(X)$.

We consider only triples $(\sim, A, \textit{lab})$ the restriction of which is compatible with the precondition constraints. This means that:

1. $\textit{mol}(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1 \le k \le n}, \sim \cap [\![1;n]\!]^2, A_{|[\![1;n]\!]}) \neq \bot_{(I(p_k))_{1 \le k \le n}}$,

2. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}((t^i_v)_{v \in \mathscr{V}_c}, \textsc{cu}(A(i))) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}},\ \forall i \in [\![1;n]\!]$,

where $t^i_v$ is the cardinal of the set $\{j \in [\![1;n]\!] \mid v = p^j,\ \textit{give-unit}(t_i) = \textit{give-unit}(t_j)\}$.

Then, we take into account the constraints denoted by *lab*. We require that the label of the value of each variable $x$ is the label $\textit{lab}(x)$. That is to say:

$$\textit{force-lab}\left(S, \textit{mol}\left(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1 \le k \le n}, \sim \cap [\![1;n]\!]^2, A_{|[\![1;n]\!]}\right)\right) \neq \bot_{(I(p^k))_{1 \le k \le n}},$$

where $S = \{(\textit{param}^k_l, k), \textit{lab}(X^k_l) \mid X^k_l \in \mathscr{V}^X_{\mathscr{R}}\}$.

We also take into account the fact that some threads are necessarily launched in new computation units: for any $(i,j)$ such that $1 \le i \le n$ and $1 \le j \le n_i$, if there exists $b \in B$ such that $\textit{give-index}(p^{(i,j)})(b) \in \textit{Dom}(E_s^{(i,j)})$, the computation unit of the $\phi(i,j)$-th thread is new. In such a case, we should have $k \not\sim n + \phi(i,j)$ for any $k$ such that $1 \leq i \leq n$.

This gives the following definition for the set of admissible constraint sets.

**Definition 10.3.4 (admissible constraint sets).** We define the set *extended-context* of admissible constraint sets as the set of triples $(\sim, A, B)$ such that:

1. $\textit{mol}(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1\leq k \leq n}, \sim \cap [\![1;n]\!]^2, A_{|[\![1;n]\!]}) = \bot_{(I(p_k))_{1\leq k\leq n}}$,
2. $\textit{force-lab}(S, \textit{mol}(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1\leq k\leq n}, \sim \cap [\![1;n]\!]^2, A_{|[\![1;n]\!]})) \neq \bot_{(I(p^k))_{1\leq k\leq n}}$, where $S = \{(\textit{param}_l^k, k), \textit{lab}(X_l^k) \mid X_l^k \in \mathscr{V}_{\mathscr{R}}^X\}$;
3. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}((t_v^i)_{v\in\mathscr{V}_c}, \textsc{cu}(A(i))) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$, $\forall i \in [\![1;n]\!]$,

   where $t_v^i$ is the cardinal of the set $\{j \in [\![1;n]\!] \mid v = p^j,\ \textit{give-unit}(t_i) = \textit{give-unit}(t_j)\}$.
4. $k \not\sim (n + \phi(i,j))$, for any $i, j, k$ such that:
   (a) $1 \leq k \leq n$,
   (b) $\exists b \in B$, $\textit{give-index}(p^{(i,j)})(b) \in \textit{Dom}(E_s^{(i,j)})$.

Then, we consider any case of potential constraint set $(\sim, A, \textit{lab}) \in$ *extended-context* for the computation units.

#### 10.3.6.3.3 Enforcing computation unit constraints

Now, we enforce the constraints given by the triple $(\sim, A, \textit{lab})$ in the balance molecule.

**Definition 10.3.5 (updated balance molecule).** For any $i$, $j$ such that $1 \leq j \leq i \leq m$, we define the abstract molecule $\textit{balance}_j^i$ by induction as:

$$\begin{cases} \textit{balance}_0^1 = \textit{balance} \\ \textit{balance}_i^i = \textit{set-unit}(i, A(i), \textit{balance}_{i-1}^i) \\ \textit{balance}_1^{i+1} = \begin{cases} \textit{same-unit}(1, i+1, (p^k), \textit{balance}_i^i) & \text{if } i+1 \sim 1, \\ \textit{distinct-unit}(1, i+1, (p^k), \textit{balance}_i^i) & \text{otherwise,} \end{cases} \\ \textit{balance}_{j+1}^i = \begin{cases} \textit{same-unit}(j+1, i, (p^k), \textit{balance}_j^i) & \text{if } j+1 \sim i, \\ \textit{distinct-unit}(j+1, i, (p^k), \textit{balance}_j^i) & \text{otherwise,} \end{cases} \end{cases}$$

We define the updated balance molecule $\textit{updated-balance}(\sim, A, \textit{lab})$ under the assumptions $(\sim, A, \textit{lab})$ as the molecule:

$$\textit{force-lab}(S, \textit{balance}_m^m),$$

where $S = \{(param_l^k, k), lab(X_l^k) \mid X_l^k \in \mathscr{V}_{\mathscr{R}}^X\}$.

**10.3.6.3.4 Launching abstract atoms** Then, we compute the abstraction of pairs marker/environment after continuation launching. For that purpose, we collect the abstraction of all the atoms of new threads in the balance molecule.

**Definition 10.3.6 (updating abstract atoms).** We define the abstract description *new-atoms* $\in \mathscr{C}_{env}^{\sharp}$ of the pairs of marker/environment of new threads as the function that maps any program point $p^{(i,j)}$ with $1 \leq i \leq n$, $1 \leq j \leq n_i$ to the following abstract atom:

$$\sqcup_{I(p^{(i,j)})}\{\mathrm{GC}^{\sharp}(I(p^{(i,j)}), \mathrm{PROJ}^{\sharp}(\phi(i',j'), \textit{updated-balance})) \mid p^{(i,j)} = p^{(i',j')}\}.$$

**10.3.6.3.5 Updating computation units** Then, we can update properties about computation units. We first count the threads that are consumed or created in the computation unit of each thread in the balance molecule.

**Definition 10.3.7.** For any $l \in [\![1; m+n]\!]$, we define:

1. the abstraction $\textit{created}(l)$ of the number of threads that are launched in the computation unit of the $l$-th thread. The abstract element $\textit{created}(l) \in \mathscr{N}_{\mathscr{V}_c}$ is defined as:
$$\sum^{\sharp}(u_k)_{n<k\leq m},$$
where: $u_k = \begin{cases} 1_{\mathscr{N}_{\mathscr{V}_c}}(p^{(i,j)}) & \text{when } \exists i, j, \begin{cases} k = \phi(i,j) \\ \text{and } k \sim l, \end{cases} \\ 0_{\mathscr{N}_{\mathscr{V}_c}} & \text{otherwise;} \end{cases}$

2. the abstraction $\textit{consumed}(l)$ of the number of threads that are consumed in the computation unit of the $l$-th thread. The abstract element $\textit{consumed}(l) \in \mathscr{N}_{\mathscr{V}_c}$ is defined as:
$$\sum^{\sharp}(v_k)_{1\leq k\leq n},$$
where: $v_k = \begin{cases} 1_{\mathscr{N}_{\mathscr{V}_c}}(p^k) & \text{if } \begin{cases} \textit{type}(pi^k) \neq \textit{replication} \\ \text{and } k \sim l; \end{cases} \\ 0_{\mathscr{N}_{\mathscr{V}_c}} & \text{otherwise;} \end{cases}$

3. the update of factious variables *transition*. The abstract element *transition* is defined as:
$$\textit{transition} = 1_{\mathscr{N}_{\mathscr{V}_c}}(\psi(\mathscr{R}, (p^k, pi^k))).$$

Then we associate each thread in the balance molecule to an abstraction of the content of the computation unit it belongs to, after continuation launching.

**Definition 10.3.8.** We define a function *new-unit* mapping each index $l \in [\![1;m+n]\!]$ to a description of the occurrence number of thread in the computation unit of the $l$-th thread, as follows:

1. if there exists $j \in [\![1;n]\!]$ such that $l \sim j$, we define *new-unit*$(l)$ as:

$$\textsc{sync}_{\mathcal{N}_{\mathcal{V}_c}}(t_j, \textsc{cu}(A(l))) -^\sharp \mathit{consumed}(l) +^\sharp \mathit{created}(l) +^\sharp \mathit{transition};$$

2. else, if $l = \phi(i,j)$ and if there exists $b \in B$ such that $A(\mathit{give\text{-}unit}(p^{(i,j)}))(b) \in \mathit{Dom}(E_s^{(i,j)})$, we define *new-unit*$(i)$ as:

$$\mathit{created}(i) +^\sharp \mathit{transition},$$

3. otherwise, we define *new-unit*$(i)$ as:

$$(0_{\mathcal{N}_{\mathcal{V}_c}} \cup \textsc{cu}(A(i))) +^\sharp \mathit{created}(i) +^\sharp \mathit{transition}(i).$$

Roughly speaking, in the first case, the fact that there exists an interacting threads in the computation unit before the computation step ensures that the computation unit was not empty; in the second case, at least one name occurring in the computation unit is fresh, so the computation unit was empty before the computation step; in the third case, we do not know whether the computation unit was empty, or not before the computation step, so we consider both cases.

*Remark* 10.3.9. When a thread is launched in a computation unit that is not constrained in the precondition, we have to consider both cases: either the computation unit is not empty, so its abstraction is given by the function $\textsc{cu}$, or the computation unit is empty, so its abstraction is given by $0_{\mathcal{N}_{\mathcal{V}_c}}$. In mobile ambients, our analysis will be less precise that the analysis we have proposed in Sect. 10.2, because when an ambient migrate inside another one. We have no constraint on the content of the target ambient. We will address this accuracy problem in future works.

**10.3.6.3.6 Primitive definitions** We summarize the primitives that we need to describe continuation launching in the abstract computation step:

**Definition 10.3.10 (continutation launching primitives).** Let $C^\sharp = (\textsc{env}, \textsc{cu}) \in \mathscr{C}^\sharp_{part}$ be an abstract configuration. Let $\mathscr{R} = (n, \mathit{components}, \mathit{compatibility}, \mathit{v\text{-}passing}, \mathit{broadcast})$ be a reduction rule.

Let $(p^k)_{1\leq k\leq n} \in \mathscr{L}_{\mathrm{p}}^n$ be an $n$-tuple of program point labels and $(pi^k)_{1\leq k\leq n} = (s^k, (param_l^k)_l, (bd_l^k)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$ be an $n$-tuple of partial interactions. Let $(V_i)_{1\leq k\leq n}$ be an $n$-tuple of interfaces (after value passing). Let $(ct^k) \in \wp(\mathscr{L}_{\mathrm{p}} \times (\mathscr{V} \rightharpoonup \mathscr{L}))$ be an $n$-tuple of continuations, such that $ct^k \in continuation^k$, for any $k$ such that $1 \leq k \leq n$. Let $mol^0 \in Molecule^{\sharp}_{(I(p^i))_{1\leq i\leq n}}$ be an abstract molecule. Let $mol' \in Molecule^{\sharp}_{(V_i)_{1\leq i\leq n}}$ be an abstract molecule. The abstract molecule $mol^0$ denotes the $n$ interacting threads just before computing value passing and marker computation; the abstract molecule $mol'$ denotes the $n$ interacting threads just after computing value passing and marker computation. We introduce, for any $i$ such that $1 \leq k \leq n$, the integer $n_i$ as the cardinal of the set $ct^i$. We also write $ct^i = \{(p^{(i,j)}, E_s^{(i,j)}) \mid 1 \leq j \leq n_i\}$. We define an integer $m$ as the sum $\Sigma_{1\leq i\leq n} n_i$. We introduce $\phi$ as the unique monotonic bijection from the set $\{(i,j) \mid 1 \leq i \leq n,\ 1 \leq j \leq n_i\}$ into the interval $[\![1;m]\!]$ (using the lexical order).

1. the set *context'*$(C^{\sharp}, \mathscr{R}, (p^k, pi^k, ct^k))$ is defined as *extended-context*;
2. the abstract element $\textsc{env}'(\mathscr{R}, (p^k, pi^k, ct^k), C^{\sharp}, mol^0, mol', \sim, A, lab)$ is defined as *new-atoms*;
3. the molecule $bal(\mathscr{R}, (p^k, pi^k, ct^k), C^{\sharp}, mol^0, mol', \sim, A, lab)$ is defined as *balance*;
4. the function $\textsc{cu}'(\mathscr{R}, (p^k, pi^k, ct^k), C^{\sharp}, \sim, A, lab)$ is defined as *new-unit*;

#### 10.3.6.4 Broadcast value passing

Broadcast value passing consists in substituting all the occurrences of a value in the system by another one. Let $C^{\sharp} = (\textsc{env}, \textsc{cu}) \in \mathscr{C}^{\sharp}_{part}$ be an abstract configuration that denotes the state of the system before the computation step. We consider a triple $(\sim, A, lab)$ that denotes constraints on the computation unit of the threads involved in the computation step. We consider *molecule* the abstract molecule that describes the interacting threads having taken into account the constraints denoted by both $\sim$ and $A$. We consider *balance* the abstract molecule that describes the interacting threads and the launched threads having taken into account the constraints denoted by $\sim$, $A$, and $lab$. We denote by $n$ the number of interacting threads. We denote by $m$ the number of launched threads. For any $k$ such that $1 \leq k \leq m+n$, we denote by $q^k$ the program point of the $k$-th thread in the abstract molecule *balance*. Let *new-atoms* $\in \mathscr{C}^{\sharp}_{env}$ be the description of the pairs marker/environment of the new threads (before broadcast communication). Let *new-unit* $\in \textsc{unit} \rightarrow \mathscr{N}_{\mathscr{V}_c}$ be the description of the computation unit of the interacting threads and of the created threads. Let $(param_l^k)_{k,l}$ be an $n$-tuple of sequences of parameters ($param_l^k$ is associated with the $l$-th parameter of the $k$-th thread). We denote by *broadcast* the broadcast substitution (it is a partial map from $\mathcal{V}_{\mathcal{R}}^{I}$ into $\mathcal{V}_{\mathcal{R}}^{X} \cup \mathcal{V}_{\mathcal{R}}^{I}$).

**10.3.6.4.1 Environment properties** We use the primitive $glob^\sharp$ which is defined in Sect. 8.1.3.5 on page 179. We first refine the abstract molecule *molecule* so that it takes into account the constraints that are denoted by the function *lab*. We define *molecule′* as the molecule:

$$\mathit{force\text{-}lab}(S, \mathit{molecule}),$$

where $S = \{(param_l^k, k), lab(X_l^k) \mid 1 \leq k \leq n,\ X_l^k \in Dom(lab)\}$.

Then, we define the abstract element:

$$\mathrm{ENV}_{res}(C^\sharp, (param_l^k), \mathit{broadcast}, \mathit{molecule}, \mathit{lab}) \in \mathscr{C}_{env}^{\sharp}$$

as the element:

$$glob^\sharp(\mathit{molecule}', (param_l^k), \mathit{broadcast}, \sqcup_{env}\{\mathrm{ENV}; \mathit{new\text{-}atoms}\}).$$

**10.3.6.4.2 Computation unit properties** At the concrete level, a restriction *broadcast′* of the function *broadcast* is chosen. Then, we compute the substitution $\tau$ as the function that maps the thread identity of the $k$-th interacting thread to the value of the variable $broadcast(I^k)$ whenever $I^k \in Dom(broadcast')$, and leaves other values unchanged. The substitution $\tau$ applies on computation units component-wise (we denote by $\overline{\tau} \in unit \to unit$ the function that applies $\tau$ component-wise). The content of each computation unit $u$, after the broadcast substitution is then obtained as the sum of the contents of all computation units $u^{-1}$ such that $\overline{\tau}(u^{-1}) = u$.

We denote by $l$ the function that maps each integer $k$ such that $1 \leq k \leq n$ to the label $l(k)$ where:

$$l(k) = \begin{cases} p^k & \text{if } I^k \notin Dom(broadcast'); \\ p^{k'} & \text{if } broadcast'(I^k) \text{ matches } I^{k'}; \\ lab(broadcast'(I^k)) & \text{otherwise.} \end{cases}$$

We introduce the family $(v_j)_{j \in J}$ of the distinct concrete computation units such that: $\{w \in unit \mid \overline{tau}(w) = u\} = \{v_j \mid j \in J\}$. In the abstract, we will only consider some constraints about each computation unit $v_j$. These constraints are described at the concrete level by the function $i \in (J \times B) \to [\![0; n]\!]$, that is defined as:

1. $i(j,b) = 0$ whenever $v_j(b)$ is not in the domain of $\tau$ (in such a case $v_j(b) = u(b)$,

2. and $i(j,b) = k$ whenever $v_j(b)$ is the identity of the $k$-th interacting thread and $I^k = Dom(broadcast')$ (in such a case $v_j(b)$ is the identity of the $k$-th interacting thread).

Moreover, for any distinct $j_1, j_2 \in J$, there exists $b \in B_s$, such that $i(j_1,b) \neq i(j_2,b)$, otherwise we would have $v_{j_1} = v_{j_2}$.

Let $u^\sharp$ be an abstract computation unit. We want to compute the set $\mathscr{F}$ of the pairs $((v_j^\sharp)_{j\in J}, i \in J \times B \rightarrow [\![0;n]\!])$ such that there exists a computation unit $u$ and a family of distinct computation units $(v_j)_{j\in J}$ that satisfy:

1. $u^\sharp = \Pi_{unit}(u)$;

2. $v^\sharp = \Pi_{unit}(v_j)$ for any $j \in J$;

3. $\{w \in unit \mid \overline{tau}(w) = u\} = \{v_j \mid j \in J\}$;

4. $i(j,b) = 0$ whenever $v_j(b)$ is not in the domain of $\tau$ (in such a case $v_j(b) = u(b)$;

5. and $i(j,b) = k$ whenever $v_j(b)$ is the identity of the $k$-th interacting thread and $I^k = Dom(broadcast')$ (in such a case $v_j(b)$ is the identity of the $k$-th interacting thread).

**Definition 10.3.11 (abstract computation unit origin).** Thus we can define $\mathscr{F}$ as the set of the family $(v_j^\sharp, i_j)_{j\in J}$ such that:

1. for any $j \in J$, $v_j^\sharp \in$ UNIT is an abstract computation unit;

2. for any $j \in J$, $i_j \in B \rightarrow [\![0;n]\!]$ is a function;

3. for any $j \in J$, $b \in B$, $i_j(b) = 0$ implies $v_j^\sharp(b) = u^\sharp(b)$;

4. for any $j \in J$, $b \in B$, $i_j(b) > 0$ implies $v^\sharp(b) = p^k$;

5. for any $j_1, j_2 \in J$, $[\forall b \in B_s,\ i_{j_1}(b) = i_{j_2}(b)] \implies j_1 = j_2$.

*Remark* 10.3.12. The last property ensures that the set $\mathscr{F}$ is finite up to re-indexing.

For any $(v^\sharp_j, i_j) \in \mathscr{F}$, the abstraction of the new content of the a concrete computation unit satisfying the abstraction $u^\sharp$ is then obtained by summing for each $j \in J$ the abstraction of the content of a computation unit that satisfies $v^\sharp_j$.

To get more precise results, we introduce a primitive *is-in-balance* that takes a family $f = (v^\sharp_j, i_j) \in \mathscr{F}$ and an index $j \in J$. It returns the set $A$ of integer $k \in [\![0; m+n]\!]$ such that for any $k \in A$ such that $k > 0$, the $k$-th thread in the balance molecule may be in a concrete computation unit that satisfies both the abstraction $v^\sharp_j$ and the constraints induced by $i_j$; and such that $0 \notin A$ only if , there may be no thread in the balance molecule in any concrete computation unit that satisfies both the abstraction $v^\sharp_j$ and the constraints induced by $i_j$.

**Definition 10.3.13.** For any $v^\sharp \in \textsc{unit}$ and any $f = (v^\sharp_j, i_j)_{i \in J} \in \mathscr{F}$, we define *is-in-balance*$(j, f)$ by:

1. $0 \notin$ *is-in-balance*$(j, f)$ if and only if:
   (a) $Card(B_s) = 1$ and
   (b) for any $k' \in [\![1; m+n]\!]$, $\textsc{sync}^\sharp(\{(give\text{-}index(q^{k'})(b), k') \neq (\sigma(i_j(b))); \mid b \in B_s\}, (q^k), balance) = \bot_{(V_i)}$,
   where $\sigma : \begin{cases} X^k_l \mapsto (param^k_l, k), \\ I^k \mapsto (I, k), \end{cases}$
2. for any $k$ such that $1 \leq k \leq m+n$, $k \in$ *is-in-balance*$(v^\sharp_j, f)$ if and only if $\textsc{sync}^\sharp(\{(give\text{-}index(q^{k'})(b), k') = (\sigma(i_j(b \mid b \in B)))\}, (q^k), balance) \neq \bot_{(V_i)}$,
   where $\sigma : \begin{cases} X^k_l \mapsto (param^k_l, k), \\ I^k \mapsto (I, k). \end{cases}$

Then, for any $k \in J$, we use the primitive *is-in-balance* to refine the abstraction of the content of the computation unit $v_k$. We consider two choices. When a computation unit contains no thread of the balance molecule, then its content is either empty, or given by the abstract precondition $\textsc{cu}(v^\sharp_k)$. When a computation unit contains a thread in the balance molecule, then its content before broadcast substitution is given by the updated abstraction *new-unit*$(v^\sharp_k)$. Thus, we define by $\phi(k, (v^\sharp_j, i_j)_{j \in J})$ the abstraction of the content of $v^\sharp_k$ as follows:

**Definition 10.3.14 (refined content).**

$$\phi(k, (v^\sharp_j, i_j)_{j \in J}) = removezero(\sqcup_{\mathscr{N}_{\mathscr{V}_c}} \textsc{old} \cup \textsc{new})$$

where:

1. $\textsc{old} = \begin{cases} \sqcup_{\mathscr{N}_{\mathscr{V}_c}}\{0_{\mathscr{N}_{\mathscr{V}_c}}; \textsc{cu}(v_k^\sharp)\} & \text{if } 0 \in \textit{is-in-balance}(v_k^\sharp, (v_j^\sharp, i_j)), \\ \perp_{\mathscr{N}_{\mathscr{V}_c}} & \text{otherwise;} \end{cases}$

2. $\textsc{new} = \begin{cases} \textit{new-unit}(v_k^\sharp) & \text{if } [\![1; m+n]\!] \cap \textit{is-in-balance}(v_k^\sharp, (v_j^\sharp, i_j)) \neq \emptyset, \\ \perp_{\mathscr{N}_{\mathscr{V}_c}} & \text{otherwise;} \end{cases}$

3. $\textit{removezero}(A) = \begin{cases} \perp_{\mathscr{N}_{\mathscr{V}_c}} & \text{if } A = 0_{\mathscr{N}_{\mathscr{V}_c}} \\ A & \text{otherwise.} \end{cases}$

Then we can update the abstraction of the content of any computation unit.

**Definition 10.3.15 (updated computation unit content).** We define the abstract element:

$$\textsc{cu}_{res}(C^\sharp, (param_l^k), broadcast, balance, lab, ct^k) \in \mathscr{C}_{env}^\sharp$$

as the element:

$$[u \mapsto \sqcup_{\mathscr{N}_{\mathscr{V}_c}}\{\sum^\sharp(\phi(j), (v_j, i_j)_{j \in J})_{j \in J} \mid J \text{is a set, } (v_j, i_j) \in \mathscr{F}\}]$$

### 10.3.6.5 Abstract operational semantics

We use these primitives in order to describe both the abstraction of the initial states and the abstract computation rule.

The abstraction of the initial states is defined as:

$$C_{0_{part}} = (C_{0_{env}}, t_0)$$

where $C_{0_{env}}$ is defined as in Sect. 8.1.4 on page 181 and $t_0$ is defined as:

$$\left[u^\sharp \mapsto \Pi_{\mathbb{N}^{\mathscr{V}_c}}\left(\left[p^k \mapsto Card\left\{(q, E_s) \in init_s \,\middle|\, \begin{array}{l} q = p^k,\ \forall b \in B, \\ E_s(\textit{give-index}(q)(b)) = u^\sharp(b) \end{array}\right\}\right]\right)\right].$$

Computation steps are described by the abstract reduction relation in Fig. 10.10. We recall the different steps of this computation, as follows:

- *interaction enabling*:
  - first, we find some threads that exhibit the right partial interactions;
  - then, we check that their interface are compatible with both formal rule synchronization constraints and each matching constraint set;

Let $C^\sharp = (f,P) \in \mathscr{C}^\sharp_{part}$ be an abstract configuration.
Let $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$ be a reduction rule.
Let $(p^k)_{1\leq k\leq n} \in \mathscr{L}_\mathrm{p}^n$ be an $n$-tuple of program point labels and $(pi^k)_{1\leq k\leq n} = (s^k, (parameter^k_l)_l, (bd^k_l)_{k,l}, constraints^k, continuation^k)_{1\leq k\leq n}$ be an $n$-tuple of partial interactions.
We denote by *context* the set of pairs $(\sim, A)$ where $\sim$ is an equivalence relation over $[\![1;n]\!]$ and $A$ a function $A \in [\![1;n]\!] \to \textsc{unit}$ such that:

1. $mol(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1\leq k\leq n}, \sim, A) \neq \bot_{(I(p_k))_{1\leq k\leq n}}$,
2. $\textsc{sync}_{\mathscr{N}_{\mathscr{V}_c}}(t_j, \textsc{cu}(A(i))) \neq \bot_{\mathscr{N}_{\mathscr{V}_c}}$, $\forall i \in [\![1;n]\!]$,

If $context \neq \emptyset$ and $\forall k \in [\![1;n]\!]$, $pi^k \in interaction(p)$, then:

$$C \overset{(\mathscr{R},(p^k,pi^k)_k)}{\longrightarrow_\sharp} \sqcup_{part} \left\{ (\overline{\textsc{env}}_{res}(c), \overline{\textsc{cu}}_{res}(c) \;\middle|\; c \in \overline{context} \times \prod_{1\leq k\leq n} continuation^k \right\}.$$

where for any $(ct^k)$ such that for any $k \in [\![1;n]\!]$, we have $ct^k \in continuation^k$, for any $(\sim, A, lab) \in \overline{context}$:

- $\overline{context} = context'(C^\sharp, \mathscr{R}, (p^k, pi^k, ct^k))$;
- $\overline{mol}(\sim, A, (ct^k)) = mol(C^\sharp, \mathscr{R}, (p^k, pi^k)_{1\leq k\leq n}, \sim \cap [\![1;n]\!]^2, A_{|[\![1;n]\!]})$;
- $\overline{mol}\,'(X) = marker\text{-}value(type(components(1)), (p^k), \overline{mol}(X), (bd^k_l), (param^k_l), v\text{-}passing)$;
- $\overline{bal}(\sim, A, lab, (ct^k)) \quad = \quad bal(\mathscr{R}, (p^k, pi^k, ct^k), C^\sharp, \overline{mol}(\sim, A, (ct^k)), \overline{mol}\,'(\sim, A, (ct^k)), \sim, A, lab)$;
- $\overline{\textsc{env}}(\sim, A, lab(ct^k)) = \textsc{env}'(\mathscr{R}, (p^k, pi^k, ct^k), C^\sharp, \sim, A, lab)$;
- $\overline{\textsc{cu}}(\sim, A, lab, (ct^k)) = \textsc{cu}'(\mathscr{R}, (p^k, pi^k, ct^k), C^\sharp, \sim, A, lab)$;
- $\overline{\textsc{env}}_{res}((\sim, A, lab), (ct^k)) = \textsc{env}_{res}(C^\sharp, (param^k_l), broadcast, m, lab)$,
  where $m = \overline{mol}(\sim, A, (ct^k))$;
- $\overline{\textsc{cu}}_{res}((\sim, A, lab), (ct^k)) = \textsc{cu}_{res}(C^\sharp, (param^k_l), broadcast, balance, lab, (ct^k))$.

Figure 10.10: Abstract semantics.

  - then, we check that each computation unit contains enough threads to compute the interaction;

- *interaction computation*:

  For each case of contextual constraints and of potential continuations:

  - we abstract marker allocation and value passing;
  - we compute the abstraction of pair of marker/environment after broadcast substitution;
  - we compute the abstraction of the content of computation unit after broadcast substitution, by case analysis.

**Theorem 10.3.16.** $(\mathscr{P}, \sqsubseteq_{\mathscr{P}}, \sqcup^{\mathscr{P}}, \bot_{\mathscr{P}}, \gamma_{\mathscr{P}}, C_0^{\mathscr{P}}, \rightsquigarrow_{\mathscr{P}}, \nabla_{\mathscr{P}})$ *is an abstraction.*

## 10.4 Applications

### 10.4.1 Race-condition analysis for the $\pi$-calculus

**Example 10.4.1 (Shared memory analysis).** *We consider the shared memory example that is given in Fig. 10.1 on page 267. We extend the interface of each program point, so that they all contain the variable* `cell`*. We suppose that threads at program points* 20 *and* 32 *associate the variable* `cell` *with the value* $(ext,\varepsilon)$*; Threads at program point* 2, 5, *and* 9 *have already the variable* `cell` *in their interface. Threads with the marker id and at program point* 3, 4, 12, *and* 16 *associate the variable* `cell` *with the value* $(c,id)$*. Other threads inherit the value of the variable* `cell` *from the the thread of which they are in the continuation. Then, we define the computation unit of a thread as the value of the variable* `cell`*.*

*The environment analysis* LOC$_2$ *(Cf. Sect. 8.2.7.2 on page 219) proves that the instances of the channels* `cell`*,* `mutex`*, or* `nomutex` *are never communicated anywhere in the system. The abstract domain of equalities and disequalities among value markers (see Sect. 8.3.1.3 on page 228) proves that the marker of the value of any variable* `cell`*,* `nomutex`*, or* `mutex` *is always the same as the computation unit marker. Moreover, the partition-based analysis proves that, in each computation unit labeled c:*

1. *the sum between the number of occurrences of threads at program points* 2, 7, *and* 11 *is always equal to one. This prove that each memory cell is associated with one and at most one value.*

2. *the sum between the number of occurrences of threads at program points* 3, , 15, *and* 19 *is always equal to one. This proves that the* mutex *policy is*

*satisfied (i.e. the mutex is either available or released and there is only one available slot).*

*Remark* 10.4.2. To prove that the client data are not sent neither to other clients, nor to the context. We need to use the analysis that is given in Sect. 8.2.11 on page 213. We require the partitioned numerical abstraction of environments to relate the values of the cell location (i.e. the value of the variable `cell`) and of the cell content (i.e. the value of variables $u$ and $d$). Nevertheless, if we remove the context (we still consider an unbounded number of clients), the analysis LOC$_2$ succeeds in proving that the client data are never sent to other clients.

## 10.4.2 Authentication analysis for the *spi*-calculus

**Example 10.4.3 (One-way public key authentication protocol analysis).** *We consider the authentication protocol given in Fig. 10.2. We extend the interface of each program point with an extra variable* unit*. We also consider a fresh name label* 0*.*

*We suppose that:*

1. *The environments of threads at program points $\alpha$, $\beta$, and $\gamma$ associate the variable* unit *to the value $(0,\varepsilon)$ (resp. $(\beta,\varepsilon)$);*

2. *The environments of threads at the program point* 1*,* 2 *and* 3 *associate the variable* unit *to the variable $(0,\varepsilon)$.*

3. *The environments of threads at the program point* 4 *associates the variable* unit *either to a secret key if this thread has been created by the reception of the corresponding public key, or to the value $(0,\varepsilon)$ otherwise.*

4. *The value of the variable* unit *is passed during computation steps to threads at program points* 5*, $c$,* 6*,* 7 *, $d$, $e$, and $f$.*

5. *The environments of threads at the program point* 8*,* 9 *and* 10 *associate the variable* unit *to the variable $(0,\varepsilon)$.*

6. *When a message (denoted by a thread $t_m$) is communicated to a thread at program point* 10*, the environment of the thread at program point* 11 *associate the variable* unit *to the value of the variable* unit *in the environment of the thread $t_m$.*

7. *The value of the variable* unit *is passed during computation steps to threads at program points* 12, 13, 14, 15, 16, 17, 18*, $g$, $h$, and $i$.*

*Remark* 10.4.4. We need to introduce new partial interactions to describe the value associated to the variable unit during communication. We include communicated terms in communication rule. Firstly, this allows us to pass the value of the variable unit of a thread denoting a term to the thread this term is communicated to. Secondly, it allows us to compute the value of the variable unit according to the shape of the communicated term.

*The environment analysis detects that:*

1. *at program point* 19*, the variable* unit *is associated with the secret key of the principal B;*

2. *if the variable* unit *at program point* 4 *is associated with the secret key of the principal B, then the argument of the* begin *signal is associated with the public key of B.*

*The analysis detects that a computation unit may contain a thread at program point* 19 *only if it contains a thread at program point* 4*. For that purposes, it makes the closure among several simpler constraints that are encoded in linear equality constraints (thanks to the use of factious variables). These simpler constraints are:*

1. *a computation unit may contain a thread at program point $d$ only if it contains a thread at program point* 4*;*

2. *a computation unit may contain a thread at program point* 12 *only if it contains a thread at program point $d$;*

3. *a computation unit may contain a thread at program point* 13 *(resp.* 14*,* 15*,* 16*,* 17*,* 18*, and* 19*) may be launched in a computation unit, only by consuming a thread at program point* 12 *(resp.* 13, 14, 15, 16, 17*, reps.* 18*) in the same computation unit.*

*This proves the non-injective agreement property: for any instance of a thread* 19*, there exists a thread at program point* 4 *such that the arguments of the two threads match.*

# Chapter 11

# Conclusion

## 11.1 Contribution

In this thesis, we propose a generic framework for analyzing mobile systems. Our approach is generic: it does not depend on the model under consideration. We give a meta language. We encode the models most frequently used in the literature, such as the $\pi$-calculus, the *ambient* calculus, the *join*-calculus, the *spi*-calculus, and the BIO-*ambients*. Thus, we model successfully matching guards, internal and external choices, guarded replication and explicit recursion, location, migration and dissolution, term construction and destruction, safe migration, and channeled communication across boundaries.

We obtain some semantics that do not use $\alpha$-conversion. Each name (channel name, ambient name, memory cell name, etc...) is tagged with a marker that encodes the local history of the thread that has declared this name. This makes further analyses easier, since the state of the system is not defined up to a congruence relation. Moreover, it allows us to express some properties about the history of the threads that declare names. For instance, given two names, we can decide whether they have been created by the same instance of a thread, or not. Moreover, our semantics are context independent. They take into account any potential intruder, that respects the semantics of the encoded models.

Then, we use the Abstract Interpretation framework to design decidable analyses. This framework is highly generic: it can be applied to a wide variety of analyses, provided some abstract primitives are given. Moreover, it is extensible: it allows us to build the (approximated reduced) product of several analyses expressed in this framework. We propose three kinds of generic analyses.

- We propose an environment analysis that associates each program point to a description of the environments that may be associated to threads at this program point. This analysis is non-uniform: it distinguishes between several instances of names. It also distinguishes between recursive instances of threads. We build a hierarchy of different environment analyses with different levels of precision (we do not compare only transfer function accuracy, but the whole analysis accuracy). Our abstract semantics succeeds in validating non-uniform confidentiality properties, such as whether the data of a client may not be communicated to other clients in the case of a *ftp*-server. We prove this property even in the case of an unbounded number of clients.

- We propose an occurrence counting analysis. The occurrence counting analysis consists in abstracting the occurrence number of thread instances during computation sequences. It is especially useful to detect mutual exclusion. It also helps in discovering a bound to the number of agents during computation sequences, so that we can verify that some part of the systems will not exceed the physical limits imposed by the implementation of the system. In the case of an *ftp*-server, we can automatically infer the maximum number of simultaneous client sessions.

- Finally, we introduce the notion of thread partitioning. Threads are gathered into computation units. These computation units are semantically defined. Then, we count the occurrence number of threads inside each computation unit. This analysis mixes both dynamic (it distinguishes among several instances of each computation unit) and concurrency properties. It succeeds in proving the absence of race-conditions in a shared memory with dynamic allocation. We also prove the *non-injective agreement* property in a version of the Woo and Lam one-way public-key authentication protocol.

## 11.2 Future works

### 11.2.1 Implementation

Our two prototypes $\pi$-s.a and *amb*-s.a have been implemented according to non standard semantics computed by hand. They address the problem of environment analysis and occurrence counting analysis. The prototype *amb*-s.a. also captures an approximation of the content of each ambient. Much works remain to do.

We want to implement a concrete interpreter for our meta language. Given a front-end file for a model, it will automatically provide a concrete interpreter for this model. We do not want to implement distributed abstract machines. Our goal is to provide quickly a tool implementation of models, in order to give intuition on the semantics of the model and to study toy examples.

Then, we want to implement all the analyzes that we have proposed in this thesis. Most of the examples that we have given in this thesis have been analyzed by using our prototypes. Nevertheless, thread partitioning results have been computed by hand. The proof of *non-injective agreement* property is quite complex, a prototype is required to check that the analysis captures it.

### 11.2.2 Proving high level properties

We capture very low level properties. These properties are required in designing accurate analyses for high level properties. We sometimes bridge the gap between the properties that we prove and the property we are interested in by manual proofs. We want to design a formal and automatic framework to do this.

Thread partitioning provides higher level properties. But it requires some kind of human annotations. We need to describe the computation unit of each thread during the computation. First, we would like to extend the meta language: we want to provide systematic tools to include new rules and new partial interactions in order to describe the behavior of computation units easily. then, we would like to propose generic families of computation units. Each generic family would address a given class of high level properties (such as the absence of race conditions). Then, the resulting analysis could be instantiated to each encoded model.

We also want to extend the class of properties that we analyze. For instance, we want to infer injective agreement properties, which ensure that at most one *end* signal is spawned per *begin*, in authentication protocols.

### 11.2.3 Extending model expressiveness

We want to extend our meta language. Our meta language is based on the use of asymmetric communication and name atomicity. We can deal neither with fusion, nor with equational theories over terms. Moreover, we think that we can encode higher order communications. But the analysis results may be quite hard to understand. Our framework may be adapted to get more useful results.

We also want to design analyses that abstract the mobile behavior of programs written in complex languages. We may need to capture interactions between properties related to mobility, and other properties of the programs (the data-flow for instance).

### 11.2.4 Approximating probabilistic behavior

In this thesis, we address only reachability properties. Many applications require more complex properties. For instance, the behavior of a cell in biology is often controlled by the concentration of the components in the whole system. The cell may either die, or duplicate itself. These two states are reachable. A reachability analysis gives no information for such a system.

Therefore, we need to analyze the probabilistic semantics of models, where each computed transition is chosen according to the concentration of the components in the system. Then, our goal is to capture the probabilistic behavior of a system.

# Appendix A

# Correspondence proves

## A.1 The standard and the naive semantics

In this section we give the proof of Thm. 2.2.5, which formalizes the relation between the standard and the naive semantics.

**Theorem 2.2.5.** *We have $\mathscr{S} = \Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}))$, and for any non standard configurations $C$ and for any word $u \in (\mathscr{L}^2 \cup \{\varepsilon;\oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) \xrightarrow{u}_{\mathrm{n}}^{*} C$, we have:*

1. $C \xrightarrow{\varepsilon}_{\mathrm{n}} C' \implies \Pi(C) \equiv \Pi(C')$;
2. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\},\ C \xrightarrow{\lambda}_{\mathrm{n}} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;
3. $\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\},\ \Pi(C) \xrightarrow{\lambda} P \implies \exists D,\ \exists E,\ \begin{cases} C \xrightarrow{\varepsilon}_{\mathrm{n}}^{*} D \xrightarrow{\lambda}_{\mathrm{n}} E \\ \Pi(E) \equiv P. \end{cases}$

*Proof.* We have $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) = (\mathscr{S}, \varepsilon, \emptyset)$, then $\Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S})) = \mathscr{S}$. Let $C$ be a non standard configuration and $u$ a word in $(\mathscr{L}^2 \cup \{\varepsilon;\oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) \xrightarrow{u}_{\mathrm{n}}^{*} C$,

1. let $C'$ be a non standard configuration such that $C \xrightarrow{\varepsilon}_{\mathrm{n}} C'$, we want to prove that $\Pi(C) \equiv \Pi(C')$ by case analysis on $C \xrightarrow{\varepsilon}_{\mathrm{n}} C'$:

   (a) in the case that $C \xrightarrow{\varepsilon}_{\mathrm{n}} C'$ consists in decomposing a thread into two concurrent threads, we have $\Pi(C) \equiv \Pi(C')$ thanks to the associativity and commutativity congruence rules;

   (b) in the case that $C \xrightarrow{\varepsilon}_{\mathrm{n}} C'$ consists in removing a thread the syntactic component of which is the empty thread, we have $\Pi(C) \equiv \Pi(C')$ thanks to the "end of a thread" congruence rule;

(c) in the case that $C \xrightarrow{\varepsilon}_{\mathrm{n}} C'$ consists in opening a channel, we have $\Pi(C) \equiv \Pi(C')$ thanks to the extrusion rule, the swapping rule, and the consistency of the marker allocation scheme (Cf. Prop. 2.2.1).

2. let $C'$ be a non standard configuration such that $C \xrightarrow{\lambda}_{\mathrm{n}} C'$, we have $\Pi(C) \xrightarrow{\lambda} \Pi(C')$: in the case that $C$ only contains the threads involved in the non standard computation step, the property is true by definition of the two relations and thanks to the fact that a standard computation step can be performed inside name restriction; then we use the fact that a standard computation step can be performed both within parallel composition and within name restriction to prove it in the general case.

3. let $P$ be a standard configuration such that $\Pi(C) \xrightarrow{\lambda} P$. The binary relation $\xrightarrow{\varepsilon}_{\mathrm{n}}$ is nœtherian and locally confluent. So, following [30], we can take a non standard configuration $D$ such that $C \xrightarrow{\varepsilon}_{\mathrm{n}}^{*} D$ and such that for any configuration $E$, $D \overset{\varepsilon}{\not\longrightarrow}_{\mathrm{n}} E$. According to Thm. 2.2.5.(1), we have $\Pi(C) \equiv \Pi(D)$, then since $\xrightarrow{\lambda}$ is compatible with $\equiv$, we get that $\Pi(D) \xrightarrow{\lambda} P$. Besides, we can deduce from the fact that $D$ cannot be reduced by a computation step labeled with $\varepsilon$, that the syntactic component of every thread of $D$ is either of the form $P \oplus Q$ or of the form $aP$. So the computation step $\Pi(D) \xrightarrow{\lambda} P$ can be lifted in the non standard semantics, so that we can choose a non standard configuration $E$ such that $D \xrightarrow{\lambda}_{\mathrm{n}} E$ and $\Pi(E) \equiv P$.

□

## A.2 The standard and the intermediate semantics

In this section we give the proof of Thm. 2.2.8, which formalizes the relation between the standard and the intermediate semantics.

**Proposition 2.2.6.** *For any non standard configuration $C$, we have $C \Downarrow \bigcup_{t \in C} \beta^{\mathrm{i}}(t)$.*

*Proof.* Since the relation $\beta^{\mathrm{i}}$ separately acts on each thread, it is enough to prove the property in the case that $C$ is a singleton $\{(P, id, E)\}$. This is done by induction on the syntax of $P$:

- in the case that $P = 0$,
  we have $\{(0,id,E)\} \xrightarrow{\varepsilon}_{\mathrm{n}} \emptyset$ and $\beta^{\mathrm{i}}(0,id,E) = \emptyset$,
  so $\{(0,id,E)\} \Downarrow \beta^{\mathrm{i}}(0,id,E)$;

- in the case that $P = (\nu\ n)Q$, we have $\{(P,id,E)\} \xrightarrow{\varepsilon}_{\mathrm{n}} \{(Q,id,E[n \mapsto (n,id)])\}$,
  and by induction $\{(Q,id,E[n \mapsto (n,id)])\} \Downarrow \beta^{\mathrm{i}}(Q,id,E[n \mapsto (n,id)])$,
  since $\beta^{\mathrm{i}}(P,id,E) = \beta^{\mathrm{i}}(Q,id,E[n \mapsto (n,id)])$,
  we obtain $\{(P,id,E)\} \Downarrow \beta^{\mathrm{i}}(P,id,E)$;

- in the case that $P = (P_1 \mid P_2)$,
  we have $\{(P_1 \mid P_2,id,E)\} \xrightarrow{\varepsilon}_{\mathrm{n}} \{(P_1,id,E_{|fn(P_1)});(P_2,id,E_{|fn(P_2)})\}$,
  and by induction $\forall i \in \{1;2\}$, $\{(P_i,id,E_{|fn(P_i)})\} \Downarrow \beta^{\mathrm{i}}(P_i,id,E_{|fn(P_i)})$,
  since[1] $\beta^{\mathrm{i}}(P_1 \mid P_2,id,E) = \beta^{\mathrm{i}}(P_1,id,E_{|fn(P_1)}) \cup \beta^{\mathrm{i}}(P_2,id,E_{|fn(P_2)})$,
  we obtain $\{(P_1 \mid P_2,id,E)\} \Downarrow \beta^{\mathrm{i}}(P_1 \mid P_2,id,E)$;

- otherwise we have $\beta^{\mathrm{i}}(P,id,E) = \{(P,id,E_{|fn(P)})\}$,
  and $\{(P,id,E_{fn(P)})\} \Downarrow \{(P,id,E_{|fn(P)})\}$,
  so $\{(P,id,E)\} \Downarrow \beta^{\mathrm{i}}(P,id,E)$.

□

**Theorem 2.2.8.** *We have $\mathscr{S} \equiv \Pi(\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}))$, and for all non standard configurations $C$ and for all word $u \in (\mathscr{L}^2 \cup \{\varepsilon;\oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) \xrightarrow{u}{}_{\mathrm{i}}^{*} C$, we have:*

1. *$\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\}$, $C \xrightarrow{\lambda}_{\mathrm{i}} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;*

2. *$\forall \lambda \in \mathscr{L}^2 \cup \{\oplus\}$, $\Pi(C) \xrightarrow{\lambda} P \implies \exists D, \begin{cases} C \xrightarrow{\lambda}_{\mathrm{i}} D \\ \Pi(D) \equiv P. \end{cases}$*

*Proof.* The proof of these properties mainly relies on Thm. 2.2.5 and the fact that $\xrightarrow{\lambda}_{\mathrm{i}}$ is defined as the composition of $\xrightarrow{\lambda}_{\mathrm{n}}$ and $\Downarrow$: accordance to Thm. 2.2.5, we know that $\mathscr{S} = \Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}))$, then by definition of $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S})$, we have $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) \xrightarrow{\varepsilon}{}_{\mathrm{n}}^{*} \mathscr{C}_0^{\mathrm{i}}(\mathscr{S})$, and so we can deduce from Thm. 2.2.5.(1) that $\mathscr{S} \equiv \Pi(\mathscr{C}_0^{\mathrm{i}}(\mathscr{S})$. Then let us take a non standard configuration $C$ and word $u \in (\mathscr{L}^2 \cup \{\varepsilon;\oplus\})^*$ such that $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) \xrightarrow{u}{}_{\mathrm{i}}^{*} C$,

1. let $C'$ be a non standard configuration such that $C \xrightarrow{\lambda}_{\mathrm{i}} C'$, there exists another non standard configuration $D$ such that $C \xrightarrow{\lambda}_{\mathrm{n}} D \xrightarrow{\varepsilon}{}_{\mathrm{n}}^{*} C'$, we

[1]We use the property that $\beta^{\mathrm{i}}(R,id,E) = \beta^{\mathrm{i}}(R,id,E_{|fn(R)})$ which can be easily proved for all threads $R$ by induction on the syntax of $R$.

obtain from Thm. 2.2.5.(2) that $\Pi(C) \xrightarrow{\lambda} \Pi(D)$ and from Thm. 2.2.5.(1) that $\Pi(D) \equiv \Pi(C')$, so $\Pi(C) \xrightarrow{\lambda} \Pi(C')$;

2. let $P$ be a standard configuration such that $\Pi(C) \xrightarrow{\lambda} P$, following Thm. 2.2.5.(3), there exists two non standard configurations $D$ and $E$, such that $C \xrightarrow{\varepsilon}_{\mathrm{n}}^{*} D$, $D \xrightarrow{\lambda}_{\mathrm{n}} E$ and $\Pi(E) \equiv P$, moreover, since $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) \xrightarrow{u}_{\mathrm{i}}^{*} C$, we know that for any non standard configuration $D'$, we have $C \overset{\varepsilon}{\not\longrightarrow}_{\mathrm{n}} D'$, this means that $D = C$, so we have $C \xrightarrow{\lambda}_{\mathrm{n}} E$ and $\Pi(E) \equiv P$, then we introduce the non standard configuration $F$ such that $E \Downarrow F$, we know from Thm. 2.2.5.(1) that $\Pi(E) \equiv \Pi(F)$, and by definition of $\xrightarrow{\lambda}_{\mathrm{i}}$ that $C \xrightarrow{\lambda}_{\mathrm{i}} F$, so we can conclude that $P \equiv \Pi(F)$ and $C \xrightarrow{\lambda}_{\mathrm{i}} F$.

$\square$

## A.3 The standard and the efficient semantics

In this section we give the proof of Thm. 2.2.10, which formalizes the relation between the standard and the efficient semantics.

**Proposition 2.2.9.** *For any non standard configuration C, we have:*

$$\{b \mid C \implies b\} = \left\{ \bigcup Cont_t \mid \forall t \in C,\ Cont_t \in \beta(t) \right\}.$$

*Proof.* Since the $\beta$ relation separately acts on each thread, it is enough to prove the property in the case that $C$ is a singleton $\{(P, id, E)\}$. This is done by induction on the syntax of $P$:

- in the case that $P = 0$, only the garbage collection rule applies to the thread $(P, id, E)$ and no rule applies on the empty set, so we conclude that $\{b \mid \{(0, id, E)\} \implies b\} = \{\emptyset\} = \beta(0, id, E)$;

- in the case that $P = (\nu\ n)Q$, only the name restriction rule can apply: we have $\{(P, id, E)\} \xrightarrow{\varepsilon}_{\mathrm{n}} \{(Q, id, E[n \mapsto (n, id)])\}$, $\{b \mid \{Q, id, E[n \mapsto (n, id)]\} \implies b\} = \beta(Q, id, E[n \mapsto (n, id)])$ (by the induction hypothesis), and $\beta(P, id, E) = \beta(Q, id, E[n \mapsto (n, id)])$,
so we obtain $\{b \mid \{(P, id, E)\} \implies b\} = \beta(P, id, E)$;

- in the case that $P = (P_1 \mid P_2)$, only the rule which decomposes the thread can apply: we have $\{(P_1 \mid P_2, id, E)\} \xrightarrow{\varepsilon}_{\mathrm{n}} \{(P_1, id, E_{|fn(P_1)}); (P_2, id, E_{|fn(P_2)})\}$, so, since $\dashrightarrow$ separately acts on each thread, we obtain $\{b \mid \{(P,id,E)\} \Longrightarrow b\} = \{b_1 \cup b_2 \mid \forall i \in \{1;2\},\ \{(P_i,id,E)\} \Longrightarrow b_i\}$; moreover by induction we know that for $i$ in the set $\{1;2\}$, we have $\{b_i \mid \{(P_i,id,E_{|fn(P_i)})\} \Longrightarrow b\} = \beta(P_i,id,E_{|fn(P_i)})$, then by definition of $\beta(P_1 \mid P_2,id,E)$, we obtain[2] $\{b \mid \{(P_1 \mid P_2,id,E)\} \Longrightarrow b\} = \beta(P_1 \mid P_2,id,E)$;

- in the case that $P = (P_1 \oplus P_2)$, only the two choice rules can apply: we have either $\{(P_1 \oplus P_2,id,E)\} \xrightarrow{\oplus}_{\mathrm{n}} \{(P_1,id,E_{|fn(P_1)})\}$ or $\{(P_1 \oplus P_2,id,E)\} \xrightarrow{\oplus}_{\mathrm{n}} \{(P_2,id,E_{|fn(P_2)})\}$, so $\{b \mid \{(P_1 \oplus P_2,id,E)\} \Longrightarrow b\} = \cup_{i\in\{1;2\}}\{b \mid \{(P_i,id,E)\}$ $\Longrightarrow b\}$; moreover by induction we know that for $i$ in the set $\{1;2\}$, we have that $\{b_i \mid \{(P_i,id,E_{|fn(P_i)})\} \Longrightarrow b\} = \beta(P_i,id,E_{|fn(P_i)})$, so since[2] $\beta(P_1 \oplus P_2,id,E) = \beta(P_1,id,E_{|fn(P_1)}) \cup \beta(P_2,id,E_{|fn(P_2)})$, we obtain $\{b \mid \{(P_1 \oplus P_2,id,E)\} \Longrightarrow b\} = \beta(P_1 \oplus P_2,id,E)$;

- otherwise we have $\beta(P,id,E) = \{\{(P,id,E_{|fn(P)})\}\}$, and $\forall b$, $\{(P,id,E_{fn(P)})\} \not\dashrightarrow b$, so $\{b \mid \{(P,id,E)\} \Longrightarrow b\} = \beta(P,id,E)$.

□

**Theorem 2.2.10.** *For any initial non standard configuration $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$, there exists $k \in \mathbb{N}$ such that $\mathscr{S} \xrightarrow{\oplus^k}{}^* \Pi(C_0)$ and for all non standard configurations $C$ and for all word $u \in (\mathscr{L}^2)^*$ such that $C_0 \xrightarrow{u}{}_{\mathrm{e}}^* C$, we have:*

1. $\forall\lambda \in \mathscr{L}^2,\ C \xrightarrow{\lambda}_{\mathrm{e}} C' \Longrightarrow \exists k \in \mathbb{N},\ \exists P,\ \Pi(C) \xrightarrow{\lambda} P \xrightarrow{\oplus^k}{}^* \Pi(C')$;

2. $\forall\lambda \in \mathscr{L}^2,\ \Pi(C) \xrightarrow{\lambda} P \Longrightarrow \exists D, \begin{cases} C \xrightarrow{\lambda}_{\mathrm{e}} D \\ \textit{and} \begin{cases} \exists k > 0,\ P \xrightarrow{\oplus^k}{}^+ \Pi(D) \\ \textit{or } P \equiv \Pi(D). \end{cases} \end{cases}$

*Proof.* Let $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$ be an initial configuration, since $\mathscr{C}_0^{\mathrm{e}}(\mathscr{S}) = \beta(\mathscr{S},\varepsilon,\emptyset)$, we obtain that $\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}) \dashrightarrow^* C_0$, then by definition of $\dashrightarrow$ and thanks to Thm. 2.2.5.(1) and Thm. 2.2.5.(2), there exists $k \in \mathbb{N}$ such that $\Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S})) \xrightarrow{\oplus^k}{}^* \Pi(C_0)$, since $\mathscr{S} = \Pi(\mathscr{C}_0^{\mathrm{n}}(\mathscr{S}))$ we get that $\mathscr{S} \xrightarrow{\oplus^k}{}^* \Pi(C_0)$. Then let us take a non standard configuration $C$ and a word $u \in (\mathscr{L}^2)^*$ such that $C_0 \xrightarrow{u}{}_{\mathrm{e}}^* C$,

[2]We use the property that $\beta(R,id,E) = \beta(R,id,E_{|fn(R)})$ which can be easily proved for all threads $R$ by induction on the syntax of $R$.

1. let $C'$ be a non standard configuration such that $C \xrightarrow{\lambda}_{\mathrm{e}} C'$, by definition of $\xrightarrow{\lambda}_{\mathrm{e}}$ , there exists another non standard configuration $D$ such that we have $C \xrightarrow{\lambda}_{\mathrm{n}} D \dashrightarrow^{*} C'$; moreover, we obtain from Thm. 2.2.5.(2) that $\Pi(C) \xrightarrow{\lambda} \Pi(D)$ and we deduce from Thm. 2.2.5.(1) and Thm. 2.2.5.(2) that there exists $k \in \mathbb{N}$, such that $\Pi(D) \xrightarrow{\oplus^k}{}^{*} \Pi(C')$;

2. let $P$ be a standard configuration such that $\Pi(C) \xrightarrow{\lambda} P$, following Thm. 2.2.5.(3), there exist two non standard configurations $D$ and $E$ which satisfy $C \xrightarrow{\varepsilon}{}_{\mathrm{n}}^{*} D$, $D \xrightarrow{\lambda}_{\mathrm{n}} E$ and $\Pi(E) \equiv P$; since $\mathscr{C}_0^{\mathrm{i}}(\mathscr{S}) \xrightarrow{u}{}_{\mathrm{e}}^{*} C$, we know that for any non standard configuration $D'$, we have $C \not\dashrightarrow D'$, this means that $D = C$ and $\lambda \neq \oplus$, so we have $C \xrightarrow{\lambda}_{\mathrm{n}} E$ and $\Pi(E) \equiv P$, then we introduce the non standard configuration $F$ such that $E \Longrightarrow F$, by definition of $\xrightarrow{\lambda}_{\mathrm{e}}$ , we have $C \xrightarrow{\lambda}_{\mathrm{e}} F$; moreover we have either $E \xrightarrow{\varepsilon}{}_{\mathrm{n}}^{*} F$ and thanks to Thm. 2.2.5.(1) $\Pi(E) \equiv \Pi(F)$, or $E \dashrightarrow^{*} E' \xrightarrow{\oplus}_{\mathrm{n}} E'' \dashrightarrow^{*} F$, and thanks to Thm. 2.2.5.(1) and Thm. 2.2.5.(2), there exists $k > 0$ such that $\Pi(E) \xrightarrow{\oplus^+}{}^{*} \Pi(F)$.

$\square$

# Appendix B

# Marker freshness

## B.1 Thread marker freshness

In this section we give the proof of Thm. 4.5.13.

We first prove that the computation of a single partial interaction may not launch two instances of threads with labels that are in the same program point class and that two instances of the same thread with labels in the same class may not occur at the beginning of the system computation:

**Lemma B.1.1.** *Let $i,k$ be two integers such that $i \in [\![0;o]\!]$ and such that either $k = 0$ if $i = 0$, or $1 \leq k \leq n_i$ otherwise. Let $p$ and $p'$ be two program points, let $id$ be a marker, and let $E$,$E'$ be two environments such that $(p,id,E) \in created(i,k)$ and $(p',id,E') \in created(i,k)$. Then, if $p \sim p'$, then $p = p'$ and $E = E'$.*

*Proof.* We prove the Lem. B.1.1 by case analysis as follows:

1. in the case when $i = k = 0$:

   Let $t = (p,id,E)$ and $t' = (p',id',E')$ be two threads in $C_0$ such that $p \sim p'$ and $id = id'$. Thanks to the definition of $\mathscr{C}_0$ there exists $Ct \in init_s$ such that $\{t;t'\} \subseteq launch(Ct,\varepsilon,\emptyset)$. Thanks to Def. 4.4.7, there exist two static environments $E_s,E'_s$ such that $t = (p,\varepsilon,update(\varepsilon,\emptyset,E_s))$ and $t' = (p',\varepsilon,update(\varepsilon,\emptyset,E'_s))$ and $(p,E_s),(p',E_s) \in Ct$. Thanks to Def. 4.5.4, since $Ct \in init_s$ and $p \sim p'$, we have $E_s = E'_s$ and $p = p'$. So we can conclude that: $E = E'$ and $p = p'$.

2. otherwise: Let $t = (p,id,E)$, $t' = (p',id',E')$ be two threads in $created(i,k)$, such that $p \sim p'$ and $id = id'$. We want to prove that $E = E'$:
   By Def. 4.4.7, there exist two static environments $E_s,E'_s$ such that:

   - $\{(p,E_s);(p',E'_s)\} \subseteq Ct^k(i)$;

- $E = update(\overline{id}^k(i), \overline{E}^k(i), E_s)$;
- $E' = update(\overline{id}^k(i), \overline{E}^k(i), E'_s)$.

By Def. 4.5.4, we have $p = p'$ and $E_s = E'_s$. So $E = E'$ and $p = p'$.

□

The following lemma allows the tracking of thread origin. It gives a characterization of the initial threads and relates each non initial thread $t$ to a thread that has been consumed when launching the thread $t$.

**Lemma B.1.2 (Parent thread).** *Let $t = (p, id, E)$ be a thread in $created(i,k)$, then:*

1. *in the case where $k = i = 0$, we have $id = \varepsilon$ and $t \in init_p$;*
2. *in the case where $type(s^k(i)) = computation$, there exist an environment $E'$ and a program point $q$ such that $t' = (q, id, E') \in consumed(i,k)$, moreover the class $[q]_\sim$ is the only class such that $[p]_\sim \in succ([q]_\sim)$;*
3. *in the case where $type(s^k(i)) = replication$, the marker $id$ matches $N((p_1, ..., p_n), id_1, ..., id_n)$, moreover for any $l \in [\![1;n]\!]$ such that $type(s^l(i)) = computation$, there exists $E'$ such that $(p_l, id_l, E') \in consumed(i,l)$;*
4. *in the case where $type(s^k(i)) = migration$, there exists a program point $q \in [p]_\sim$, an environment $E'$ such that $t' = (q, id, E') \in consumed(i,k)$;*

This way, a regular computation may launch a thread only if it consumes a thread the program point of which is in the syntactic predecessor of the created thread and the marker of which is the same. When a replication launches an instance, the associated marker allows the tracking of the thread which are computed (thanks to Def. 4.1.6 there are always at least one consumed thread). In case of migration, the thread is obtained by updating the environment of an existing thread and the program point:

**Lemma B.1.3.** *Let $i$ and $k$ be two integers such that $type(s^k(i)) = migration$. Then there exists two program points $p$ and $p'$, a marker $id$ and two environments $E$ and $E'$ such that: $p \sim p'$, $consumed(i,k) = \{(p, id, E)\}$ and $created(i,k) = \{(p', id, E')\}$.*

We now prove our main theorem:

**Theorem 4.5.13 (Thread marker freshness).** *Let $i$ be an integer between $0$ and $o$ and let $(p,id,E)$ be a thread in $C_i$. Then, there exists an unique 4-tuple $(p',i',k,E')$ such that $p'$ is a program point label, $i'$ and $k$ are integers, and $E'$ is an environment that satisfy $p \sim p'$, $(p',id,E') \in created(i',k)$ and such that either $i = 0$, or $type(s^k(i)) \neq migration$.*

We prove Thm. 4.5.13 by proving the following stronger theorem:

**Theorem B.1.4.** *Let $i$ be an integer between $0$ and $o$. Then:*

1. *let $(p,id,E)$ be a thread in $C_i$. Then there exists an unique 4-tuple $(p',i',k,E')$ such that $p'$ is a program point label, $i'$ and $k$ are integers, and $E'$ is an environment that satisfy $p \sim p'$, $(p',id,E') \in created(i',k)$ and either $i = 0$, or $type(s^k(i)) \neq migration$.*
2. *Let $p$ and $p'$ be two program points and let $id$ be a marker. Let $E$ and $E'$ be two environments such that $(p,id,E) \in C_i$ and $(p',id,E') \in C_i$. If $p \sim p'$ then $p = p'$ and $E = E'$.*

*Proof.* The proof is made by induction over $i$:

1. in the case where $i = 0$. By definition, we have $C_0 = created(0,0)$. The two points follows from Lem. B.1.1.
2. we suppose that there exists an integer $i_0$ such that Thm. B.1.4 is satisfied for any $i < i_0$; we prove that it is also valid for $i = i_0$:

   (a) Let $p_1$ and $p_2$ be two program point labels and $id$ be a marker. Let $E_1$, $E_2$ be two environments, and $i_1$, $i_2$, $k_1$ and $k_2$ be four integers such that:

   - $p_1 \sim p_2$,
   - $(p_1,id,E_1) \in created(i_1,k_1)$,
   - $(p_2,id,E_2) \in created(i_2,k_2)$,
   - $type(s^{k_1}(i_1)) \neq migration$,
   - and $type(s^{k_2}(i_2)) \neq migration$.

   We make a case analysis:

   i. in the case where $type(s^{k_1}(i_1)) = computation$:
   By Lemma B.1.2, there exist a program point $q_1$ and an environment $E'_1$ such that $(q_1,id,E'_1) \in consumed(i_1,k_1)$ and $[p_1]_\sim \in succ([q_1]_\sim)$. So we have $(q_1,id,E'_1) \in C_{i_1-1}$. By Lemma 4.5.12, there exist a program point $q'_1$, two integers $j_1$ and $l_1$, and an environment $E''_1$ such that $(q'_1,id,E''_1) \in created(j_1,l_1)$,

$j_1 < i_1$, $type(s^{l_1}(j_1)) \neq migration$, and $q_1 \sim q'_1$. By Def. 4.5.8, we have $type(s^{k_2}(i_2)) = computation$. So by Lemma B.1.2, there exist a program point $q_2$ an environment $E'_2$ such that $(q_2, id, E'_2) \in consumed(i_2, k_2)$ and $[p_2]_\sim \in succ([q_2]_\sim)$. So we have $(q_2, id, E'_2) \in C_{i_2-1}$ By Lemma 4.5.12, there exist a program point $q'_2$, two integers $j_2$ and $l_2$, and an environment $E''_2$ such that $(q'_2, id, E''_2) \in created(j_2, l_2)$, $j_2 < i_2$, $type(s^{l_1}(j_1)) \neq migration$, and $q_2 \sim q'_2$. We have $p_1 \sim p_2$, $[p_1]_\sim \in succ([q_1]_\sim)$, and $[p_2]_\sim \in succ([q_2]_\sim)$. So, by Def. 4.5.8, we have $q_2 \sim q_1$. Then, $q'_2 \sim q'_1$. So by induction hypotheses, we have $q'_2 = q'_1$, $j_1 = j_2$, $l_1 = l_2$, and $E''_1 = E''_2$.
We suppose that $i_1 < i_2$. Since $type(s^{k_1}(i_1)) = computation$ and thanks to the second induction hypothesis, there is no environment $E$ and no program point $r_1$ such that $(r_1, id, E) \in C_{i_1}$ and $q_1 \sim r_1$. Moreover for any $i' \geq i_1$, there is no environment $E$ and no program point $r_1$ such that $(r_1, id, E) \in C_{i'}$ and $q_1 \sim r_1$, otherwise there would exists $(r'_1, i', k', E)$ such that $i' > i_1$, $(r'_1, id, E) \in created(i', k')$, and $r_1 \sim r'_1$, which is absurd thanks to the first induction hypothesis. This is absurd since $(q_2, id, E'_2) \in C_{i_2-1}$. So we can conclude that $i_1 = i_2$. Then we can conclude by the first induction hypothesis that $E'_1 = E'_2$ and $q_1 = q_2$. Then we have $k_1 = k_2$ because at each computation step, interacting threads are distinct pair wise. We can also conclude that $E = E'$ and $p_1 = p_2$ by lemma B.1.1.

ii. in the case where $type(s^{k_1}(i_1)) = replication$:
By Lemma B.1.2, the marker $id$ matches $N((o_1, \ldots, o_n), id_1, \ldots, id_n)$. By Def. 4.5.8, we have $type(s^{k_2}(i_2)) = replication$. By Def. 4.1.6, $type(s^2(i_1)) = computation$. So by Lemma B.1.2, there exists $E'_1$, such that $(o_2, id_2, E'_1) \in consumed(i_1, 2)$. We have $(o_2, id_2, E'_1) \in C_{i_1-1}$. By Lemma 4.5.12, there exist a program point $o'_2$, two integers $j_1$ and $l_1$, and an environment $E''_1$ such that $(o'_2, id_2, E''_1) \in created(j_1, l_1)$, $j_1 < i_1$, $type(s^{l_1}(j_1)) \neq migration$, and $o_2 \sim o'_2$. By Def. 4.1.6, we have $type(s^2(i_2)) = computation$. So by Lemma B.1.2, there exists $E'_2$ such that $(o_2, id_2, E'_2) \in consumed(i_2, 2)$. So we have $(o_2, id_2, E'_2) \in C_{i_2-1}$. By Lemma 4.5.12, there exist a program point $o''_2$, two integers $j_2$ and $l_2$, and an environment $E''_2$ such that $(o''_2, id_2, E''_2) \in created(j_2, l_2)$, $j_2 < i_2$, $type(s^{l_2}(j_2)) \neq migration$, and $o_2 \sim o''_2$. We have: $o'_2 \sim o_2 \sim o''_2$. So by induction hypotheses, we have $o'_2 = o''_2$, $j_1 = j_2$, $l_1 = l_2$, and $E''_1 = E''_2$.

We suppose that $i_1 < i_2$. Since $type(s^2(i_1)) = computation$ and thanks to the second induction hypothesis, there is no environment $E$ and no program point $r_1$ such that $(r_1, id_2, E) \in C_{i_1}$ and $o_2 \sim r_1$. Moreover for any $i' \geq i_1$, there is no environment $E$ and no program point $r_1$ such that $(r_1, id_2, E) \in C_{i'}$ and $o_2 \sim r_1$, otherwise there would exists $(r'_1, i'', k'', E)$ such that $i'' > i_1$, $(r'_1, id, E) \in created(i'', k'')$, and $r_1 \sim r'_1$, which is absurd thanks to the first induction hypothesis. This is absurd since $(o_2, id_2, E'_2) \in C_{i_2-1}$. So we can conclude that $i_1 = i_2$. Then we can conclude by the first induction hypothesis that $E'_1 = E'_2$. Then, we have $k_1 = k_2$ because at each computation step, interacting threads are distinct pair wise. We can also conclude that $E = E'$ and $p_1 = p_2$, by lemma B.1.1.

(b) We suppose that the first point is satisfied for any $i$ such that $i \leq i_0$. Let $(p, id, E)$ be a thread in $C_{i_0}$. We make a case analysis:

i. if there exists $p', i, k, E'$ such that $(p', id, E') \in created(i, k)$, $type(s^k(i)) \neq migration$, and $p' \sim p$:
By the first point of the Thm. B.1.4, we have $((p'', id, E'') \in created(i', k')$, $type(s^{k'}(i')) \neq migration$, and $p'' \sim p)$ implies that $p'' = p'$, $i = i'$, $k = k'$ and $E' = E''$. We can conclude because broadcast communications modify neither program points, nor thread markers.

ii. Otherwise, there exists $E''$ and $p'$ such that $(p', id, E'') \in C_{i-1}$ and $p \sim p'$. By the first induction hypothesis, there is no triple $(p'', k', F)$ such that $(p'', id, F) \in created(i, k')$, $type(s^{k'}(i)) \neq migration$ and $p'' \sim p$. We can conclude since both broadcast communications modify neither program points, nor thread markers and migration preserves equalities among program points.

$\square$

# Appendix C

# Context approximation

## C.1 In the $\pi$-calculus

In this section we give the proof of Thms. 6.2.8 and 6.2.9, which formalize the relation between the semantics of closed and open systems.

### C.1.1 Trace projection

We first recall Def. 6.2.7:

**Definition 6.2.7. (trace projection).** Computation sequence projection is then defined as follows: Let $\tau = C_0 \xrightarrow{\lambda_1}_{\mathrm{e}} \dots \xrightarrow{\lambda_n}_{\mathrm{e}} C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$. We define the projection of $\tau$, $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau)$ as the non standard computation sequence:

$$(A_0, U_0, F_0) \overset{\Pi_\lambda(\lambda_{a_1})}{\looparrowright} \dots \overset{\Pi_\lambda(\lambda_{a_p})}{\looparrowright} (A_p, U_p, F_p)$$

of the open system $\mathscr{S}_{\mathrm{I}}$, where

- $a_1, \dots, a_p$ is the strictly ascending sequence of the elements of the set $\{i \in [\![1;n]\!] \mid \lambda_i \in \mathscr{L}^2 \setminus (\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}})^2\}$;
- the initial configuration $(A_0, U_0, F_0)$ is the following triple:

$$(\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_0), en, \{t_n \mid n \in \mathbb{N}\});$$

- for $k \in [\![1;p]\!]$, the configuration $(A_k, U_k, F_k)$ is defined as follows:
  - $A_k = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_{a_k})$,

– $U_k = \begin{cases} U_{k-1} & \text{if } \mathit{fst}(\lambda_{a_k}) \in \mathscr{L}_{\mathrm{I}}, \\ U_{k-1} \cup \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(x_r)) \mid r \in [\![1;n]\!]\} & \text{otherwise,} \end{cases}$
where, in the last case, $(x!^j[x_1,\ldots,x_n]P, id, E)$ is the unique thread in $C_{a_{k-1}} \setminus C_{a_k}$ which matches this notation;

– $F_k = \begin{cases} F_{k-1} \text{ if } \begin{cases} \mathit{snd}(\lambda_{a_k}) \in \mathscr{L}_{\mathrm{I}} \text{ or} \\ \xrightarrow{\lambda_{a_k}}_{\mathrm{e}} \text{ is not a resource fetching,} \end{cases} \\ F_{k-1} \setminus \{\Phi_{\mathscr{M}}(\mathit{snd}(\lambda_{a_k}), id) \mid (P, id, E) \in C_{a_k-1} \setminus C_{a_k}\} \text{ otherwise.} \end{cases}$

## C.1.2 Soundness

Then, we introduce some lemma to help proving the soundness of the context independent semantics.

The following lemma shows that the extraction function $\beta$ and the configuration projection commute:

**Lemma C.1.1.** *Let $(P, id, E)$ be a thread such that $P$ is a sub-term of $\mathscr{S}_{\mathrm{I}}$, then we have:*

$$\beta(P, \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id), [x \mapsto \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(x))]) = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(\beta(P, id, E)).$$

*Proof.* This lemma can easily by proved by induction on the syntax of $P$. We use the fact that each sub-term of a sub-term of $\mathscr{S}_{\mathrm{I}}$ is also a sub-term of $\mathscr{S}_{\mathrm{I}}$. □ □

The following lemma establishes the fact that communications between two threads of the context leave the threads of the system $\mathscr{S}_{\mathrm{I}}$ unchanged:

**Lemma C.1.2.** *Let $\tau = C_0 \xrightarrow{\lambda_1}_{\mathrm{e}} \ldots \xrightarrow{\lambda_n}_{\mathrm{e}} C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$; we denote by $(A_0, U_0, F_0)\ldots(A_p, U_p, F_p)$ the computation sequence $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau)$, then we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_p$.*

*Proof.* We prove Lem. C.1.2 by induction over the integer $n - a_p$.

1. in the case that $n = a_p$, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_p$ by Def. 6.2.7;

2. we now suppose that there exists $m_0 \in \mathbb{N}$ such that Lem. C.1.2 is satisfied provided that $n - a_p < m_0$, we now prove it in the case that $n - a_p = m_0$: we know from the induction hypothesis that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n-1}) = A_p$, then we have $\lambda_n \in (\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}})^2$, so we can conclude that both the set $C_{n-1} \setminus C_n$ and the set $C_n \setminus C_{n-1}$ only contain threads the label of which is in $\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}}$, thus with respect with Def. 6.2.7, we obtain that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n-1})$, since $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n-1}) = A_p$, we conclude that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_p$. □

□

The following lemma establishes some soundness conditions:

**Lemma C.1.3.** *Let $\tau = C_0 \xrightarrow{\lambda_1}_{\mathrm{e}} \dots \xrightarrow{\lambda_n}_{\mathrm{e}} C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$. We consider the following computation sequence:*

$$\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = (A_0, U_0, F_0)\dots(A_{a_p}, U_{a_p}, F_{a_p}).$$

*Then, $\forall (P, id, E) \in C_n$ with $lab(P) \notin \mathscr{L}_{\mathrm{I}}$:*

*1. $\forall x \in fn(P)$, we have $\Phi_{\mathscr{N}}(E(x)) \in U_p$;*

*2. $\Phi_{\mathscr{M}}(lab(P), id) \in F_p$.*

*Proof.* These two properties are easily proved by induction on $n$.

1. (a) in the case that $n = 0$, we have $p = 0$; let $(P, id, E)$ be a thread in $C_0$ with $lab(P) \notin \mathscr{L}_{\mathrm{I}}$; we have $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$, let $x$ be a free name in $P$; since $lab(P) \notin \mathscr{L}_{\mathrm{I}}$, we have $x \notin \mathscr{N}_{\mathrm{I}}$; then $E(x) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(x, \varepsilon) = \Phi_{\mathscr{N}}(x, \varepsilon) \in en = U_0$;

   (b) we suppose that there exists $n_0 \in \mathbb{N}$ such that Prop. C.1.3.(1) is satisfied for any $n$ smaller than $n_0$. We now prove this property for $n = n_0 + 1$: let $(P, id, E)$ in $C_{n_0+1}$ with $lab(P) \notin \mathscr{L}_{\mathrm{I}}$, and $x$ be a free name in $P$; we denote $(y, id_y) = E(x)$;

   - in the case that there exists $j$ smaller than $n_0$ such that $(P, id, E)$ in $C_j$: by the induction hypothesis, there exists $o$ smaller than $p$ such that $\Phi_{\mathscr{N}}(E(x)) \in U_o \subseteq U_p$;
   - in the case that $y \notin \mathscr{N}_{\mathrm{I}}$, we have $\Phi_{\mathscr{N}}(E(x)) \in en = U_0 \subseteq U_p$;
   - otherwise we necessarily have $\lambda_{n_0+1} \in ((\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}}) \times \mathscr{L}_{\mathrm{I}})$, and there exists a thread $t = (z!^j[z_1, \dots, z_n]P, id_t, E_t) \in C_{n_0} \setminus C_{n_0+1}$ such that $E(x) \in \{E_t(z_l) \mid l \in [\![1; n]\!]\}$, so by definition of $U_p$, we get that $\Phi_{\mathscr{N}}(E(x)) \in U_p$.

2. (a) in the case that $n = 0$, we have $p = 0$ and $F_p = \{t_n \mid n \in \mathbb{N}\}$, so $\Phi_{\mathscr{M}}(\{(lab(P), id) \mid (P, id, E) \in C_0,\ lab(P) \notin \mathscr{L}_{\mathrm{I}}\}) \subseteq F_p$;

   (b) we suppose that there exists $n_0 \in \mathbb{N}$ such that Prop. C.1.3.(2) is satisfied for any $n$ smaller than $n_0$. We now prove this property for $n = n_0 + 1$: let $(P, id, E)$ in $C_{n_0+1}$ with $lab(P) \notin \mathscr{L}_{\mathrm{I}}$;

   - in the case that $(P, id, E) \in C_{n_0}$ and $\lambda_{n_0+1} \in (\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}})^2$: we have $\Phi_{\mathscr{M}}(lab(P), id) \in F_p$, by the induction hypothesis;

- in the case that $(P, id, E) \in C_{n_0}$ and $\lambda_{n_0+1} \notin (\mathscr{L} \setminus \mathscr{L}_{\mathrm{I}})^2$: we know by the induction hypothesis that $\Phi_{\mathscr{M}}(lab(P), id) \in F_{p-1}$, then the marker $\Phi_{\mathscr{M}}(lab(P), id)$ is also in $F_p$, otherwise we would have $(P, id, E) \in C_{n_0} \setminus C_{n_0+1}$, which is absurd;
- in the case that $(P, id, E) \notin C_{n_0}$, we have $\Phi_{\mathscr{M}}(lab(P), id) \in F_p$, otherwise there would exist an integer $i < p$ such that $\Phi_{\mathscr{M}}(lab(P), id) \in F_i \setminus F_{i+1}$, and so there would exist an integer $j < n_0 + 1$ such that $(P, id, E) \in C_j \setminus C_{j+1}$ which is in contradiction with the fact that $(P, id, E) \in C_{n_0+1}$ thanks to Prop. 2.2.1.

So in any case, we have $\Phi_{\mathscr{M}}(lab(P), id) \in F_p$. □

□

**Theorem 6.2.8. (Soundness)** *Let $\tau = C_0 \ldots C_n$ be a non standard computation sequence of the following closed system:*

$$\mathscr{S} = (\nu\, c_1) \ldots (\nu\, c_k)(\mathscr{S}_{\mathrm{I}}(c_{i_1}, \ldots, c_{i_n}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1}, \ldots, c_{j_l})),$$

*with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$. Then $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = (A_0, U_0, F_0) \ldots (A_p, U_p, F_p)$ is a non standard computation sequence of the open system $\mathscr{S}_{\mathrm{I}}$ and $(A_0, U_0, F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S})$.*

*Proof.* Soundness is ensured by construction: we prove Thm. 6.2.8 by induction on the length of the computation sequence:

1. First, we prove that $(A_0, U_0, F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S})$: we have $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$; so $C_0 \in \beta(\mathscr{S}, \varepsilon, \emptyset)$; so by definition of $\beta$, we have $C_0 \in \{A \cup B \mid A \in \beta(\mathscr{S}_{\mathrm{I}}, \varepsilon, [c_{i_k} \mapsto (c_{i_k}, \varepsilon)]),\ B \in \beta(\mathscr{S}_{\mathrm{c}}, \varepsilon, [c_{j_k} \mapsto (c_{j_k}, \varepsilon)])\}$; then we decompose $C_0$ into $A \cup B$ with $A \in \beta(\mathscr{S}_{\mathrm{I}}, \varepsilon, [c_{i_k} \mapsto (c_{i_k}, \varepsilon)])$ and with $B \in \beta(\mathscr{S}_{\mathrm{c}}, \varepsilon, [c_{j_k} \mapsto (c_{j_k}, \varepsilon)])$. Moreover, we have $A_0 = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_0) = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(A) \cup \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(B)$; since $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(B) = \emptyset$, we have $A_0 = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(A)$. Thanks to Lem. C.1.1, since $A \in \beta(\mathscr{S}_{\mathrm{I}}, \varepsilon, [c_{i_k} \mapsto (c_{i_k}, \varepsilon)])$, we obtain $A_0 \in \beta(\mathscr{S}_{\mathrm{I}}, \varepsilon, [c_{i_k} \mapsto \Phi_{\mathscr{N}}(c_{i_k}, \varepsilon)])$; so, since $\Phi_{\mathscr{N}} : (\mathscr{N} \times \mathscr{M} \to en)$, we obtain the fact that $A_0 \in \bigcup\{\beta(\mathscr{S}_{\mathrm{I}}, \varepsilon, E) \mid E \in fn(P) \to en\}$; then since $U_0 = en$ and $F_0 = \{t_n \mid n \in \mathbb{N}\}$, we obtain, by definition of $\mathscr{C}_0^{\mathrm{o}}$, that $(A_0, U_0, F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S})$.

2. Now, we assume that Thm. 6.2.8 is satisfied for any non standard computation sequence $\tau$ containing at most $n$ computation steps, and we prove that it is also satisfied for any non standard computation sequence $\tau$ of containing $n+1$ computation steps: let $\tau = C_0 \ldots C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}$ be a non standard computation sequence of length $n+1$; by the induction hypothesis

$\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(C_0\ldots C_n)$ is a non standard computation sequence of the open system $\mathscr{S}_\mathrm{I}$, which we may denote:

$$(A_0,U_0,F_0)\ldots(A_{a_p},U_{a_p},F_{a_p})$$

then we discuss several cases depending of $\lambda=(i,j)$:

(a) in the case that $\lambda\in(\mathscr{L}\setminus\mathscr{L}_\mathrm{I})^2$: by definition of $\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})$, $\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(\tau) = \Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(C_0\ldots C_n)$; so $\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(\tau)$ is a non standard computation sequence of the open system $\mathscr{S}_\mathrm{I}$;

(b) in the case that $\lambda\in\mathscr{L}_\mathrm{I}^2$ and $\xrightarrow{\lambda}_\mathrm{e}$ is a communication rule: there exist two threads $t_? = (y?^i[\overline{y}]P,id_?,E_?)$ and $t_! = (x!^j[\overline{x}],id_!,E_!)$ in $C_n$ which satisfy that $E_?(y)=E_!(x)$ and $\lambda=(i,j)$, and two continuations $Cont_?\in\beta(P,id_?,E_?[y_i\mapsto x_i])$ and $Cont_!\in\beta(Q,id_!,E_!)$ such that $C_{n+1}=(C_n\setminus\{t_?;t_!\})\cup Cont_?\cup Cont_!$; we have:

$$\begin{cases}(\Pi_\lambda(t_?),\Pi_\lambda(t_!))\in(A_p)^2 \text{ (thanks to Lem. C.1.2 and since } (i,j)\in\mathscr{L}_\mathrm{I}^2),\\ \Pi_\mathscr{N}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(E_?(y)) = \Pi_\mathscr{N}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(E_!(x)) \text{ (since } E_?(y)=E_!(x)),\\ \Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(Cont_?)\in\beta(P,\Pi_\mathscr{M}^{\Phi_\mathscr{M}}(id_?),E'_?) \text{ (thanks to Lem. C.1.1)},\\ \qquad \text{where } E'_? = [x\mapsto\Pi_\mathscr{N}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(E_?[y_k\mapsto E_!(x_k)](x))]\\ \Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(Cont_!)\in\beta(Q,\Pi_\mathscr{M}^{\Phi_\mathscr{M}}(id_!),[x\mapsto\Pi_\mathscr{N}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(E_!(x))]),\\ \qquad\qquad\qquad\qquad \text{(thanks to Lem. C.1.1);}\end{cases}$$

so $(A_p,U_p,F_p) \overset{\lambda}{\looparrowright} (A_p \setminus \{\Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(t_?);\Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(t_!)\} \cup \Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(Cont_?)\cup\Pi_\mathrm{t}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(Cont_!),U_p,F_p)$; then since $\Pi_\lambda(\lambda)=\lambda$, we may conclude that $(A_p,U_p,F_p) \overset{\Pi_\lambda(\lambda)}{\looparrowright} (\Pi_\mathrm{C}^{\Phi_\mathscr{M},\Phi_\mathscr{N}}(C_{n+1}),U_p,F_p)$;

(c) in the case that $\lambda\in\mathscr{L}_\mathrm{I}^2$ and $\xrightarrow{\lambda}_\mathrm{e}$ is a replication rule: there exist two threads $t_? = (*y?^i[\overline{y}]P,id_?,E_?)$ and $t_! = (x!^j[\overline{x}],id_!,E_!)$ in $C_n$ which satisfy that $E_?(y)=E_!(x)$ and $\lambda=(i,j)$, and two continuations $Cont_?\in\beta(P,N((i,j),id_?,id_!),E_?[y_i\mapsto x_i])$ and $Cont_!\in\beta(Q,id_!,E_!)$ such that $C_{n+1}=(C_n\setminus\{t_!\})\cup Cont_?\cup Cont_!$; since $(i,j)\in\mathscr{L}_\mathrm{I}^2$, we have $\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(N((i,j),id_?,id_!)) = N((i,j),\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_?),\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_!))$; then:

$$\begin{cases} (\Pi_\lambda(t_?),\Pi_\lambda(t_!)) \in (A_p)^2 \text{ (thanks to Lem. C.1.2 and since } (i,j)\in\mathscr{L}_{\mathrm{I}}^2\text{)},\\ \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_?(y)) = \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_!(x)) \text{ (since } E_?(y)=E_!(x)\text{)},\\ \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_?) \in \beta(P,id_*,E_*) \text{ (thanks to Lem. C.1.1)},\\ \qquad \text{where } id_* = N((i,j),\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_?),\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_!))\\ \qquad \text{and } E_* = [x\mapsto \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_?[y_k\mapsto E_!(x_k)](x))],\\ \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_!) \in \beta(Q,\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_!),[x\mapsto \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_!(x))])\\ \qquad\qquad\qquad\qquad \text{(thanks to Lem. C.1.1);} \end{cases}$$

so we have $(A_p,U_p,F_p) \overset{\lambda}{\hookrightarrow} (A_p \setminus \{\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(t_!)\} \cup \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_?) \cup \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_!),U_p,F_p)$; then since $\Pi_\lambda(\lambda)=\lambda$, we may conclude that $(A_p,U_p,F_p) \overset{\Pi_\lambda(\lambda)}{\hookrightarrow} (\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{C}}(C_{n+1}),U_p,F_p)$;

(d) in the case that $i\in\mathscr{L}_{\mathrm{I}}$, $j\notin\mathscr{L}_{\mathrm{I}}$ and $\xrightarrow{\lambda}_{\mathrm{e}}$ is a communication rule: there exist two threads $t_? = (y?^i[\overline{y}]P,id_?,E_?)$ and $t_! = (x!^j[\overline{x}],id_!,E_!)$ in $C_n$ which satisfy that $E_?(y)=E_!(x)$ and $\lambda=(i,j)$, and two continuations $Cont_? \in \beta(P,id_?,E_?[y_i\mapsto x_i])$ and $Cont_! \in \beta(Q,id_!,E_!)$ such that $C_{n+1} = (C_n\setminus\{t_?;t_!\})\cup Cont_?\cup Cont_!$; we have:

$$\begin{cases} \Pi_\lambda(t_?) \in A_p \text{ (thanks to Lem. C.1.2 and since } i\in\mathscr{L}_{\mathrm{I}}\text{)},\\ \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_?(y)) = \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_!(x)) \in U_p \text{ (since } E_?(y)=E_!(x),\\ \qquad\qquad\qquad\qquad \text{and thanks to Lem. C.1.3.(1))},\\ \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_?) \in \beta(P,\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(id_?),E'_?) \text{ (thanks to Lem. C.1.1)},\\ \qquad \text{where } E'_? = [x\mapsto \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_?[y_k\mapsto E_!(x_k)](x))]\\ \{\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E_!(x_k))\} \subseteq U_p \text{ (thanks to Lem. C.1.3.(1));} \end{cases}$$

so $(A_p,U_p,F_p) \overset{(i,0)}{\hookrightarrow} (A_p\setminus\{\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(t_?)\}\cup\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(Cont_?),U_p,F_p)$; since $\Pi_\lambda(\lambda)=(i,0)$, $(A_p,U_p,F_p) \overset{\Pi_\lambda(\lambda)}{\hookrightarrow} (\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{C}}(C_{n+1}),U_p,F_p)$;

(e) in the case that $i\in\mathscr{L}_{\mathrm{I}}$, $j\notin\mathscr{L}_{\mathrm{I}}$ and $\xrightarrow{\lambda}_{\mathrm{e}}$ is a replication rule: there exist two threads $t_? = (*y?^i[\overline{y}]P,id_?,E_?)$ and $t_! = (x!^j[\overline{x}],id_!,E_!)$ in $C_n$ which satisfy that $E_?(y)=E_!(x)$ and $\lambda=(i,j)$, and two continuations $Cont_? \in \beta(P,N((i,j),id_?,id_!),E_?[y_i\mapsto x_i])$ and $Cont_! \in \beta(Q,id_!,E_!)$

such that $C_{n+1} = (C_n \setminus \{t_!\}) \cup Cont_? \cup Cont_!$; we have:

$$\begin{cases} \Pi_\lambda(t_?) \in A_p \text{ (thanks to Lem. C.1.2 and since } i \in \mathscr{L}_\mathrm{I}\text{)},\\ \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_?(y)) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x)) \in U_p \text{ (since } E_?(y) = E_!(x) \\ \qquad\qquad \text{and thanks to Lem. C.1.3.(1))},\\ \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_?) \in \beta(P, id_*, E_*), \text{ (thanks to Lem. C.1.1)},\\ \qquad \text{where } id_* = N((i,0), \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_?), \Phi_{\mathscr{M}}(j, id_!))\\ \qquad \text{and } E_* = [x \mapsto \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_?[y_k \mapsto E_!(x_k)](x))],\\ \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x_k))\} \subseteq U_p \text{ (thanks to Lem. C.1.3.(1))},\\ \Phi_{\mathscr{M}}(j, id_!) \in F_p \text{ (thanks to the Lem. C.1.3.(2))}; \end{cases}$$

so $(A_p, U_p, F_p) \overset{(i,0)}{\looparrowright} (A_p \cup \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_?), U_p, F_p \setminus \{\Phi_{\mathscr{M}}(j, id_!)\})$; then since we gave $\Pi_\lambda(\lambda) = (i,0)$, we may conclude that $(A_p, U_p, F_p) \overset{\Pi_\lambda(\lambda)}{\looparrowright} (\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+1}), U_p, F_p \setminus \{\Phi_{\mathscr{M}}(j, id_!)\})$;

(f) in the case that $i \notin \mathscr{L}_\mathrm{I}$, $j \in \mathscr{L}_\mathrm{I}$ and $\xrightarrow{\lambda}_{\mathrm{e}}$ is a communication rule: there exist two threads $t_? = (y?^i[\overline{y}]P, id_?, E_?)$ and $t_! = (x!^j[\overline{x}], id_!, E_!)$ in $C_n$ which satisfy that $E_?(y) = E_!(x)$ and $\lambda = (i, j)$, and two continuations $Cont_? \in \beta(P, id_?, E_?[y_i \mapsto x_i])$ and $Cont_! \in \beta(Q, id_!, E_!)$ such that $C_{n+1} = (C_n \setminus \{t_?; t_!\}) \cup Cont_? \cup Cont_!$; we have:

$$\begin{cases} \Pi_\lambda(t_!) \in A_p \text{ (thanks to Lem. C.1.2 and since } j \in \mathscr{L}_\mathrm{I}\text{)},\\ \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(y)) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_?(x)) \in U_p \text{ (since } E_?(y) = E_!(x) \\ \qquad\qquad \text{and thanks to Lem. C.1.3.(1))},\\ \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_!) \in \beta(Q, \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_!), [x \mapsto \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x))),\\ \qquad\qquad \text{(thanks to Lem. C.1.1)}; \end{cases}$$

so $(A_p, U_p, F_p) \overset{(0,j)}{\looparrowright} (A_p \setminus \{\Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(t_!)\} \cup \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_!), U_p \cup \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x_k))\}, F_p)$; since $\Pi_\lambda(\lambda) = (0, j)$, we may conclude that:

$$(A_p, U_p, F_p) \overset{\Pi_\lambda(\lambda)}{\looparrowright} (\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+1}), U_p \cup \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x_k))\}, F_p);$$

(g) in the case that $i \notin \mathscr{L}_\mathrm{I}$, $j \in \mathscr{L}_\mathrm{I}$ and $\xrightarrow{\lambda}_{\mathrm{e}}$ is a replication rule: there exist two threads $t_? = (*y?^i[\overline{y}]P, id_?, E_?)$ and $t_! = (x!^j[\overline{x}], id_!, E_!)$ in $C_n$ which satisfy that $E_?(y) = E_!(x)$ and $\lambda = (i, j)$, and two continuations $Cont_? \in \beta(P, N((i, j), id_?, id_!), E_?[y_i \mapsto x_i])$ and $Cont_! \in \beta(Q, id_!, E_!)$

such that $C_{n+1} = (C_n \setminus \{t_?; t_!\}) \cup Cont_? \cup Cont_!$; we have:

$$\begin{cases} \Pi_\lambda(t_!) \in A_p \text{ (thanks to Lem. C.1.2 and since } j \in \mathscr{L}_{\mathrm{I}}), \\ \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(y)) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_?(x)) \in U_p, \text{ (since } E_?(y) = E_!(x) \\ \qquad\qquad\qquad\qquad \text{and thanks to Lem. C.1.3.(1)),} \\ \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_!) \in \beta(Q, \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_!), [x \mapsto \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x))]), \\ \qquad\qquad\qquad\qquad \text{(thanks to Lem. C.1.1);} \end{cases}$$

so $(A_p, U_p, F_p) \overset{(0,j)}{\looparrowright} (A_p \setminus \{\Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(t_!)\} \cup \Pi_{\mathrm{t}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont_!), U_p \cup \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x_k))\}, F_p)$; since $\Pi_\lambda(\lambda) = (0, j)$, we get $(A_p, U_p, F_p) \overset{\Pi_\lambda(\lambda)}{\looparrowright} (A_{p+1}, U_p \cup \{\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_!(x_k))\}, F_p)$.

$\square$

### C.1.3 Completeness

Then, we introduce a lemma to help proving the completeness of the context independent semantics.

**Lemma C.1.4.** *Let $\mathscr{S}_*$ be a closed system of the following form:*

$$(\nu\, \overline{c})(\mathscr{S}_{\mathrm{I}}(c_{i_1}, ..., c_{i_p}) \mid \mathscr{S}_{\mathrm{c}}(c_{j_1}, ..., c_{j_q})),$$

*we denote by $\mathscr{L}_{\mathrm{I}}$ the set of the label occurring in $\mathscr{S}_{\mathrm{I}}$. Let $\tau = C_0 \longrightarrow_{\mathrm{e}}^* C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S})$ and $\Phi_{\mathscr{M}} : \mathscr{L} \times \mathscr{M} \to \{t_n \mid n \in \mathbb{N}\}$ and $\Phi_{\mathscr{N}} : \mathscr{N} \times \mathscr{M} \to$ en be two one-to-one maps, then:*

1. *for any marker id occurring in a state of the computation sequence $\tau$ such that id matches $N((i, j), id_?, id_!)$, we have:*
   - $i \in \mathscr{L}_{\mathrm{I}} \implies \begin{cases} \textit{either } id_? = \varepsilon, \\ \textit{or } id_? \textit{ matches } N((i', j'), id'_?, id'_!) \textit{ where } i' \in \mathscr{L}_{\mathrm{I}}; \end{cases}$
   - $j \in \mathscr{L}_{\mathrm{I}} \implies \begin{cases} \textit{either } id_! = \varepsilon, \\ \textit{or } id_! \textit{ matches } N((i', j'), id'_?, id'_!) \textit{ where } i' \in \mathscr{L}_{\mathrm{I}}; \end{cases}$
2. *let $id_1$ and $id_2$ be two markers occurring in a state of the computation sequence $\tau$ such that $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_1) = \Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id_2)$, then $id_1 = id_2$;*
3. *let $c_1$ and $c_2$ be two names in $\mathscr{N} \times \mathscr{M}$ occurring in a state of the computation sequence $\tau$ such that $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(c_1) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(c_2)$, then $c_1 = c_2$.*

*Proof.* 1. Prop. (1) can be proved using both the fact that:

   (a) if the label $i$ of a thread $t$ is in the set $\mathscr{L}_\mathrm{I}$, then either its marker is $\varepsilon$, or there exists a marker $i'$ such that the syntactic agent labeled with $i'$ contains the sub-term labeled with $i$, and that the marker of the thread $t$ is necessarily of the form $N((i',j),id_?,id_!)$ where $i' \in \mathscr{L}_\mathrm{I}$,

   (b) a marker of the form $N((i,j),id_?,id_!)$ can be created only when a thread the label of which is $i$ and the marker of which is $id_?$ interacts with a thread the label of which is $j$ and the marker of which is $id_!$;

2. Prop. (2) can be proved by induction over the height of the markers; this induction only uses Prop. (1) and the fact that the subtree of a marker occurring in a state of $\tau$ necessarily occurs in a previous state of $\tau$;

3. Prop. (3) follows from Prop. (2) and the fact that whenever a channel name has been opened by a restriction in $\mathscr{S}_\mathrm{I}$, its marker is also the marker of a former thread the label of which is in $\mathscr{L}_\mathrm{I}$.

□

**Theorem 6.2.9. (Completeness)** *Let $\tau'$ be the non standard computation sequence of an open system $\mathscr{S}_\mathrm{I}$, that we denote by:*

$$(C_0,U_0,F_0) \overset{(i_1,j_1)}{\looparrowright} \dots \overset{(i_n,j_n)}{\looparrowright} (C_n,U_n,F_n),$$

*where $(C_0,U_0,F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S}_\mathrm{I})$.*

*Then, there exists:*

- *a closed system $\mathscr{S}_* = (\nu\,\overline{c})(\mathscr{S}_\mathrm{I}(c_{i_1},\dots,c_{i_n}) \mid \mathscr{S}_\mathrm{c}(c_{j_1},\dots,c_{j_l}))$,*
- *two one-to-one functions $\Phi_\mathscr{N}$ and $\Phi_\mathscr{M}$,*
- *a non standard computation sequence $\tau$ of the system $\mathscr{S}_*$,*

*such that $\Pi_\tau(\mathscr{S}_\mathrm{I},\Phi_\mathscr{M},\Phi_\mathscr{N})(\tau) = \tau'$.*

We will prove the following stronger result:

**Proposition C.1.5.** *Let $\tau'$ be the non standard computation sequence of an open system $\mathscr{S}_\mathrm{I}$, defined as follows:*

$$(A_0,U_0,F_0) \overset{(i_1,j_1)}{\looparrowright} \dots \overset{(i_n,j_n)}{\looparrowright} (A_n,U_n,F_n),$$

*where $(A_0,U_0,F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S}_\mathrm{I})$. Let $\mathscr{L}_\mathrm{I} \subseteq \mathscr{L}$ be the subset of the labels occurring in $\mathscr{S}_\mathrm{I}$ and $\mathscr{N}_\mathrm{I} \subseteq \mathscr{N}$ be the subset of the names occurring in name restrictions of $\mathscr{S}_\mathrm{I}$.*

*Then there exists:*

- *a closed system of the form:*

$$\begin{array}{l}\mathcal{S}_* = (\nu\,\mathrm{unsafe})(\nu\,x_1)\ldots(\nu\,x_p)\\ (\mathrm{unsafe}![x_1] \mid \ldots \mid \mathrm{unsafe}![x_p] \mid \mathcal{S}_{\mathrm{I}}(x_{i_1},\ldots,x_{i_n}) \mid \mathcal{S}_{\mathrm{c}}(\mathrm{unsafe}))\end{array}$$

  *where*

$$\begin{array}{ll}\mathcal{S}_{\mathrm{c}} = & (\nu\,\mathrm{new})\\ & (\ \boldsymbol{new} \mid \boldsymbol{repli}\\ & \mid \boldsymbol{spy}_0 \mid \ldots \mid \boldsymbol{spy}_n\\ & \mid \boldsymbol{spoil}_0 \mid \ldots \mid \boldsymbol{spoil}_n\\ & \mid \mathrm{new}![]\\ & )\end{array}$$

  *and*

  - $\boldsymbol{new} := *\mathrm{new}?[]((\nu\,channel)(\mathrm{unsafe}![channel] \mid \mathrm{new}![]))$
  - $\boldsymbol{repli} := *\mathrm{unsafe}?[x](\mathrm{unsafe}![x] \mid \mathrm{unsafe}![x])$
  - $\boldsymbol{spy}_i := *\mathrm{unsafe}?[c]c?[y_1,\ldots,y_i](\mathrm{unsafe}![y_1] \mid \ldots \mid \mathrm{unsafe}![y_i])$
  - $\boldsymbol{spoil}_i := *\mathrm{unsafe}?[c]\mathrm{unsafe}?[x_1]\ldots\mathrm{unsafe}?[x_i]c![x_1,\ldots,x_i]$

- *a non standard computation sequence $\tau$ of the closed system $\mathcal{S}_*$,*

- *two into maps $\Phi_{\mathcal{N}}$ and $\Phi_{\mathcal{M}}$, such that:*

  - $\Phi_{\mathcal{N}} : \left\{ (y, id_y) \;\middle|\; \begin{array}{l} y \notin \mathcal{N}_{\mathrm{I}},\ \exists (P, id, E) \in \bigcup C_i,\ lab(P) \in \mathcal{L}_{\mathrm{I}},\\ \exists x \in fn(P),\ E(x) = (y, id_y)\end{array} \right\} \to en;$
  - $\Phi_{\mathcal{M}} : \left\{ id \;\middle|\; \begin{array}{l} \exists (P, N((i,0), id_?, id_!), E) \in \bigcup C_i,\\ lab(P) \in \mathcal{L}_{\mathrm{I}}\end{array} \right\} \to \{t_n \mid n \in \mathbb{N}\}.$

*such that:*

1. $\Pi_\tau(\mathcal{S}_{\mathrm{I}}, \Phi_{\mathcal{M}}, \Phi_{\mathcal{N}})(\tau) = \tau'$[1];
2. *for any unsafe name $(x, id_x) \in U_n$, that occurs in a state of $\tau'$, there exists a thread in the last state of $\tau$ of the form $(\mathrm{unsafe}![y], id, E)$ with $\Pi_{\mathcal{N}}^{\Phi_{\mathcal{M}},\Phi_{\mathcal{N}}}(E(y)) = (x, id_x)$.*

[1]we make abusively no distinction neither between $\Phi_{\mathcal{N}}$ and any one-to-one extension of it defined over the set $\mathcal{N} \times \mathcal{M}$, nor between $\Phi_{\mathcal{M}}$ and any one-to-one extension of it defined over the set $\mathcal{M}$

*Proof.* We prove Prop. C.1.5 by induction on the length of the computation sequence $\tau'$:

1. in the case that $\tau'$ is of the form $(A_0,U_0,F_0) \in \mathscr{C}_0^{\mathrm{o}}(\mathscr{S}_{\mathrm{I}})$, there exists $E \in fn(\mathscr{S}_{\mathrm{I}}) \to en$ such that $A_0 \in \beta(\mathscr{S}_{\mathrm{I}},\varepsilon,E)$; we also have $U_0 = en$ and $F_0 = \{t_n \mid n \in \mathbb{N}\}$;
   we take $\Phi_{\mathscr{M}} : \emptyset \to \{t_n \mid \mathbb{N}\}$; we denote by $e_1,\ldots,e_u$ the elements of the set $\{E(x) \mid x \in fn(\mathscr{S}_{\mathrm{I}})\}$, we take $\Phi_{\mathscr{N}} : \{c_{i_k} \mid k \in [\![1;u]\!]\} \to en$, such that for any $k \in [\![1;u]\!]$, we have $\Phi_{\mathscr{N}}(c_{i_k}) = e_i$; thanks to Lem. C.1.1, we have $\beta(\mathscr{S}_{\mathrm{I}},\varepsilon,E) = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(\beta(\mathscr{S}_*,\varepsilon,\Phi_{\mathscr{N}}^{-1} \circ E))$; so we may choose $C_0 \in \beta(\mathscr{S}_*,\varepsilon,\Phi_{\mathscr{N}}^{-1} \circ E)$ such that $A_0 = \Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_0)$;

   - we have $(A_0,U_0,F_0) = \Pi_\tau(\mathscr{S}_{\mathrm{I}},\Phi_{\mathscr{M}},\Phi_{\mathscr{N}})(C_0)$;
   - for any unsafe name $(x,id_x) \in U_0$ occurring in $A_0$, there exists by construction a thread of the form $(\mathrm{unsafe}![y],id,E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = (x,id_x)$;

2. in the case that $\tau'$ is of the form $(A_0,U_0,F_0)\ldots(A_{p-1},U_{p-1},F_{p-1}) \overset{\lambda}{\looparrowright} (A_p,U_p,F_p)$; we assume by the induction hypothesis that there exist:

   - a non standard computation sequence $\tau = C_0 \ldots C_n$ of the closed system $\mathscr{S}_*$ such that $C_0 \in \mathscr{C}_0^{\mathrm{e}}(\mathscr{S}_*)$,
   - two into maps $\Phi_{\mathscr{N}}$ and $\Phi_{\mathscr{M}}$, such that:
     - $\Phi_{\mathscr{N}} : \left\{ (y,id_y) \;\middle|\; \begin{array}{l} y \notin \mathscr{N}_{\mathrm{I}},\ \exists (P,id,E) \in \bigcup C_i, \\ lab(P) \in \mathscr{L}_{\mathrm{I}}, \\ \exists x \in fn(P),\ E(x) = (y,id_y) \end{array} \right\} \to en,$
     - $\Phi_{\mathscr{M}} : \left\{ id \;\middle|\; \begin{array}{l} \exists (P,N((i,0),id_?,id_!),E) \in \bigcup C_i, \\ lab(P) \in \mathscr{L}_{\mathrm{I}} \end{array} \right\} \to \{t_n \mid n \in \mathbb{N}\};$

   such that:

   - $\Pi_\tau(\mathscr{S}_{\mathrm{I}},\Phi_{\mathscr{M}},\Phi_{\mathscr{N}})(\tau) = (A_0,U_0,F_0)\ldots(A_{p-1},U_{p-1},F_{p-1})$;
   - for any unsafe name $(x,id_x) \in U_n$ that occurs in a state of $\tau'$, there exists a thread in the last state of $\tau$ of the form $(\mathrm{unsafe}![y],id,E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = (x,id_x)$.

   We proceed by case analysis according to the kind of the next computation step:

   (a) in the case that $(A_{p-1},U_{p-1},F_{p-1}) \overset{\lambda}{\looparrowright} (A_p,U_p,F_p)$ is a communication rule: there exist two threads $t_? = (y?^i[\overline{y}]P,id_?,E_?)$ and $t_! =$

$(x!^j[\overline{x}]Q, id_!, E_!)$ with $E_?(y) = E_!(x)$, two continuations $Cont_? \in \beta(P, id_?, E_?[y_i \mapsto E_!(x_i)])$ and $Cont_! \in \beta(Q, id_!, E_!)$ such that $A_p = (A_{p-1} \setminus \{t_?; t_!\}) \cup Cont_? \cup Cont_!$; moreover, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_n) = A_{p-1}$, so by definition of $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}$ the state $C_n$ contains two threads $t'_? = (y?^i[\overline{y}]P, id'_?, E'_?)$ and $t'_! = (x!^j[\overline{x}]Q, id'_!, E'_!)$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_?) = id_?$, $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_!) = id_!$, $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}} \circ E'_? = E_?$ and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}} \circ E'_! = E_!$; thanks to Lem. C.1.1, there also exist $Cont'_? \in \beta(P, id'_?, E'_?[y_k \mapsto E'_!(x_k)])$ and $Cont'_! \in \beta(Q, id'_!, E'_!)$ such that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(Cont'_?) = Cont_?$ and $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(Cont'_!) = Cont_!$;
we set $C_{n+1} = (C_n \setminus \{t'_?; t'_!\}) \cup Cont'_? \cup Cont'_!$;

- thanks to Lem. C.1.4.(3), since $E_?(y) = E_!(x)$, we have $E'_?(y) = E'_!(x)$; so we have $C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}$; moreover $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_{n+1}) = A_p$; so $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(C_0 \dots C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}) = \tau'$;
- the last computation step of $\tau'$ does not involve unsafe name and the computation step $C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}$ does not consume any thread of the form $(\mathrm{unsafe}![y], id, E)$; so by the induction hypothesis, we get that for any unsafe name $(x, id_x) \in U_n$ occurring in a state of $\tau'$, there exists a thread in the last state of $\tau$ of the form $(\mathrm{unsafe}![y], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(E(y)) = (x, id_x)$;

(b) in the case that $(A_{p-1}, U_{p-1}, F_{p-1}) \overset{\lambda}{\looparrowright} (A_p, U_p, F_p)$ is a resource replication; there are two threads $t_? = (*y?^i[\overline{y}]P, id_?, E_?)$ and $t_! = (x!^j[\overline{x}]Q, id_!, E_!)$ with $E_?(y) = E_!(x)$, two continuations $Cont_? \in \beta(P, N((i,j), id_?, id_!), E_?[y_i \mapsto E_!(x_i)])$ and $Cont_! \in \beta(Q, id_!, E_!)$ such that $A_p = (A_{p-1} \setminus \{t_!\}) \cup Cont_? \cup Cont_!$; moreover, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_n) = A_{p-1}$, so by definition of $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}$ there exist two threads $t'_? = (*y?^i[\overline{y}]P, id'_?, E'_?)$ and $t'_! = (x!^j[\overline{x}]Q, id'_!, E'_!)$ in $C_n$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_?) = id_?$, $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_!) = id_!$, $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}} \circ E'_? = E_?$ and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}} \circ E'_! = E_!$; moreover we have $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(N((i,j), id'_?, id'_!)) = N((i,j), id_?, id_!)$; thanks to Lem. C.1.1, there also exist two continuations $Cont'_! \in \beta(Q, id'_!, E'_!)$ and $Cont'_? \in \beta(P, N((i,j), id'_?, id'_!), E'_?[y_k \mapsto E'_!(x_k)])$ such that the properties $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(Cont'_?) = Cont_?$ and $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(Cont'_!) = Cont_!$ are satisfied;
we set $C_{n+1} = (C_n \setminus \{t'_!\}) \cup Cont'_? \cup Cont'_!$:

- thanks to Lem. C.1.4.(3), since $E_?(y) = E_!(x)$, we have $E'_?(y) = E'_!(x)$; so we have $C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}$; moreover $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_{n+1}) = A_p$; so $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(C_0 \dots C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}) = \tau'$;

- the last computation step of $\tau'$ does not involve unsafe name and the computation step $C_n \xrightarrow{\lambda}_{\mathrm{e}} C_{n+1}$ does not consume any thread of the form $(\mathrm{unsafe}![y], id, E)$; so by the induction hypothesis, we get that for any unsafe name $(x, id_x) \in U_n$ occurring in a state of $\tau'$, there exists a thread in the last state of $\tau$ of the form $(\mathrm{unsafe}![y], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = (x, id_x)$;

(c) in the case that $(A_{p-1}, U_{p-1}, F_{p-1}) \overset{\lambda}{\looparrowright} (A_p, U_p, F_p)$ is a spied communication: there is a thread $t_! = (x!^j[x_1, \ldots, x_k]Q, id_!, E_!)$ with $E_!(x) \in U_{p-1}$, a continuation $Cont_! \in \beta(Q, id_!, E_!)$ such that $A_p = (A_{p-1} \setminus \{t_!\}) \cup Cont_!$, and $U_p = U_{p-1} \cup \{E_!(x_i) \mid i \in [\![1;k]\!]\}$; moreover, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_{p-1}$, so there exists a thread $t'_! = (x!^j[\overline{x}]Q, id'_!, E'_!)$ in $C_n$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_!) = id_!$ and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}} \circ E'_! = E_!$; thanks to Lem. C.1.1, there also exists $Cont'_! \in \beta(Q, id'_!, E'_!)$ such that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont'_!) = Cont_!$; thanks to the induction hypothesis, there exists a thread $t_c = (\mathit{unsafe}![c], id_c, E_c)$ in $C_n$, such that $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_c(c)) = E_!(x)$; we first use the resource **repli** to replicate the thread $t_c$: we obtain a configuration $C_{n+1} = C_n \setminus \{t_c\} \cup \{t'_c; t''_c\}$ where $t'_c$ matches $(\mathit{unsafe}![c], id'_c, E'_c)$ and $t''_c$ matches $(\mathit{unsafe}![c], id''_c, E''_c)$ with $E'_c(c) = E''_c(c) = E_c(c)$; we then replicate the resource $\mathbf{spy}_k$ by consuming the thread $t'_c$, and obtain the configuration $C_{n+2} = C_{n+1} \setminus \{t'_c\} \cup \{t_s\}$, where $t_s$ matches $(c?[y_1, \ldots, y_k](\mathit{unsafe}![y_1] \mid \ldots \mid \mathit{unsafe}![y_k]), id_s, E_s)$ with $E_s(c) = E_c(c)$; we have $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_s(c)) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E'_!(x))$; so thanks to Lem. C.1.4.(3), we deduce that $E_s(c) = E'_!(x)$; thus we perform the communication between the thread $t'_!$ and $t_s$ to obtain the configuration $C_{n+3} = (C_{n+2} \setminus \{t_s; t'_!\}) \cup Cont'_! \cup \{t'_{u_i} \mid i \in [\![1;k]\!]\}$, where each $t'_{u_i}$ matches $(\mathit{unsafe}![y_i], id'_i, E'_i)$ with $E'_i(y_i) = E'_!(x_i)$.

- the computation sequence $C_n \longrightarrow_{\mathrm{e}}^{*} C_{n+2}$ does not involve any thread of $\mathscr{S}_{\mathrm{I}}$; by Lem. C.1.2, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+2}) = A_{p-1}$; we conclude that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+3}) = A_p$; moreover $C_{n+2} \xrightarrow{i',j}_{\mathrm{e}} C_{n+3}$ with $i' \notin \mathscr{L}_{\mathrm{I}}$; so $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(C_0 \ldots C_{n+2} \xrightarrow{(i',j)}_{\mathrm{e}} C_{n+3}) = \tau'$.
- let $(u, id_u)$ be an unsafe name occurring in a state of $\tau$:
  - in the case that $(u, id_u) = E_!(x)$, $t''_c$ is a thread of $C_{n+3}$ and matches $(\mathrm{unsafe}![y], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = E_!(x)$;
  - in the case that $(u, id_u) = E_!(x_i)$, $t_{u_i}$ is a thread of $C_{n+3}$ and matches $(\mathrm{unsafe}![y_i], id, E)$ which satisfies $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y_i)) = \Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E'_!(x_i))$; so $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y_i)) = E_!(x_i)$;

– otherwise there is necessarily a thread in $C_n$ which matches $(\text{unsafe}![y], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = (x, id_x)$ and this thread is still in $C_{n+3}$.

(d) in the case that $(A_{p-1}, U_{p-1}, F_{p-1}) \overset{\lambda}{\hookrightarrow} (A_p, U_p, F_p)$ is a spoiled communication: there is a thread $t_? = (y?^i[y_1, \ldots, y_k]P, id_?, E_?)$ with $E_?(y) \in U_{p-1}$ (we set $(u_0, id_0) = E_?(y)$), there is a channel name $(u_i, id_i) \in U_{p-1}$ for each $i \in [\![1;k]\!]$, and a continuation $Cont_? \in \beta(P, id_?, E_?[y_i \mapsto (u_i, id_i)])$ such that $A_p = (A_{p-1} \setminus t_?) \cup Cont_?$, and $U_p = U_{p-1}$; moreover, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_{p-1}$, so there exists a thread $t'_? = (y?^j[\overline{y}]P, id'_?, E'_?)$ in $C_n$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_?) = id_?$ and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}} \circ E'_? = E_?$; we first deal with the names that the context sends into the system $\mathscr{S}_{\mathrm{I}}$ for the first time: for each element $(u_i, id_i)$ of the set $\{(u_i, id_i) \mid i \in [\![1;k]\!]\}$ that does not occur in any state of $\tau$, we create a new unsafe name using the resource **new**, as the result we get a fresh name $(channel, id_{chan_i})$ and, since $(u_i, id_i)$ is necessarily in *en*, we extend the definition of $\Phi_{\mathscr{N}}$ with $\Phi_{\mathscr{N}}(channel, id_{chan_i}) = (u_i, id_i)$; we obtain a state $C_{n+1}$ and an updated into map $\Phi_{\mathscr{N}}$, such that there exists a thread $t_{c_i} = (unsafe![c], id_{c_i}, E_{c_i})$ in the state $C_{n+1}$ such that $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_{c_i}(c)) = (u_i, id_i)$, for each $i \in [\![0;k]\!]$; we then use the resource **repli** to replicate the threads $t_{c_i}$ until we obtain a configuration $C_{n+2} = (C_{n+1} \setminus \{t_{c_i} \mid i \in [\![0;k]\!]\}) \cup \{t'_{c_i} \mid i \in [\![0;k]\!]\} \cup \{t''_{c_i} \mid 0 \in [\![1;k]\!]\}$ where $t'_{c_i}$ matches $(unsafe![c], id'_{c_i}, E'_{c_i})$ and $t''_{c_i}$ matches $(unsafe![c], id''_{c_i}, E''_{c_i})$ with $E'_{c_i}(c) = E''_{c_i}(c) = E_{c_i}(c)$ such that the threads $t'_{c_i}$ and $t''_{c_i}$ are all distinct from each other; we then replicate the resource $\mathbf{spoil}_k$ by consuming the thread $t'_{c_0}$, and obtain the configuration $C_{n+3} = (C_{n+2} \setminus \{t'_{c_0}\}) \cup \{t_s\}$, where $t_s$ matches $(unsafe?[x_1] \ldots unsafe?[x_k]c![\overline{x}], id_s, E_s)$ with $E_s(c) = E_{c_0}(c)$; we then use successively the threads $t'_{c_i}$ to obtain a configuration $C_{n+4} = (C_{n+2} \setminus \{t'_{c_i} \mid i \in [\![0;k]\!]\}) \cup \{t_m\}$, where $t_m$ matches $(x_0![\overline{x}], id_m, E_m)$ with $E_m(x_i) = E''_{c_i}(c)$, for each $i \in [\![0;k]\!]$; moreover the thread $t'_? = (y?^j[\overline{y}]P, id'_?, E'_?)$ is in $C_{n+4}$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_?) = id_?$, $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}} \circ E'_? = E_?$, and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E''_{c_i}(c)) = (u_i, id_i)$, for each $i \in [\![1;k]\!]$; thanks to Lem. C.1.1, there also exists $Cont'_? \in \beta(P, id'_?, E'_?[y_i \mapsto E''_{c_i}(c)])$ such that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(Cont'_?) = Cont_?$; thus we perform the communication between the thread $t'_?$ and $t_m$ to obtain the configuration $C_{n+5} = C_{n+4} \setminus \{t'_?; t_m\} \cup Cont'_?$.

- the computation sequence $C_n \longrightarrow_{\mathrm{e}}^{*} C_{n+4}$ does not involve thread of $\mathscr{S}_{\mathrm{I}}$, by Lem. C.1.2, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+4}) = A_{p-1}$; then we conclude that $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_{n+5}) = A_p$; moreover $C_{n+4} \xrightarrow{i,j'}_{\mathrm{e}} C_{n+5}$ with $j' \notin \mathscr{L}_{\mathrm{I}}$; so $\Pi_\tau(\mathscr{S}_{\mathrm{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(C_0 \ldots C_{n+4} \xrightarrow{(i,j')}_{\mathrm{e}} C_{n+5}) = \tau'$.

- let $(u, id_u)$ be an unsafe name occurring in a state of $\tau$:
  - in the case that $(u, id_u) = (u_i, id_i)$ with $i \in [\![0;k]\!]$, $t''_{c_i}$ is a thread of $C_{n+5}$ and matches $(\mathrm{unsafe}![c], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(c)) = (u_i, id_i)$;
  - otherwise there is necessarily a thread in $C_n$ which matches $(\mathrm{unsafe}![y], id, E)$ with $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E(y)) = (x, id_x)$ and this thread is still in $C_{n+5}$.

(e) in the case that $(A_{p-1}, U_{p-1}, F_{p-1}) \overset{\lambda}{\hookrightarrow} (A_p, U_p, F_p)$ is a spoiled replication: there is a thread $t_? = (*y?^i[y_1, \ldots, y_k]P, id_?, E_?)$ with $E_?(y) \in U_{p-1}$ (we set $(u_0, id_0) = E_?(y)$), there are several channel names $(u_i, id_i) \in U_{p-1}$, for $i \in [\![1;k]\!]$, a marker $id_! \in F_{p_1}$ and a continuation $Cont_? \in \beta(P, N((i,0), id_?, id_!), E_?[y_i \mapsto (u_i, id_i)])$, such that $A_p = A_{p-1} \cup Cont_!$, $U_p = U_{p-1}$ and $F_p = F_{p-1} \setminus \{id_!\}$; moreover, we have $\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(C_n) = A_{p-1}$, so there exists a thread $t'_? = (*y?^j[\overline{y}]P, id'_?, E'_?)$ in $C_n$ with $\Pi_{\mathscr{M}}^{\Phi_{\mathscr{M}}}(id'_?) = id_?$ and $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}} \circ E'_? = E_?$; we first deal with the names that the context sends into the system $\mathscr{S}_{\mathrm{I}}$ for the first time: for each element $(u_i, id_i)$ of the set $\{(u_i, id_i) \mid i \in [\![1;k]\!]\}$ that does not occur in any state of $\tau$, we create a new unsafe name using the resource **new**, as the result we get a fresh name $(channel, id_{chan_i})$ and, since $(u_i, id_i)$ is necessarily in $en$, we extend the definition of $\Phi_{\mathscr{N}}$ with $\Phi_{\mathscr{N}}(channel, id_{chan_i}) = (u_i, id_i)$; we obtain a state $C_{n+1}$ and an updated into map $\Phi_{\mathscr{N}}$, such that there exists a thread $t_{c_i} = (unsafe![c], id_{c_i}, E_{c_i})$ in the state $C_{n+1}$ such that $\Pi_{\mathscr{N}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(E_{c_i}(c)) = (u_i, id_i)$, for each $i \in [\![0;k]\!]$; we then use the resource **repli** to replicate the threads $t_{c_i}$ until we obtain a configuration $C_{n+2} = (C_{n+1} \setminus \{t_{c_i} \mid i \in [\![0;k]\!]\}) \cup \{t'_{c_i} \mid i \in [\![0;k]\!]\} \cup \{t''_{c_i} \mid 0 \in [\![1;k]\!]\}$ where $t'_{c_i}$ matches $(unsafe![c], id\text{'}_{c_i}, E'_{c_i})$ and $t''_{c_i}$ matches $(unsafe![c], id\text{''}_{c_i}, E''_{c_i})$ with $E'_{c_i}(c) = E''_{c_i}(c) = E_{c_i}(c)$, such that the threads $t'_{c_i}$ and $t''_{c_i}$ are all distinct from each other; we then replicate the resource **spoil**$_k$ by consuming the thread $t'_{c_0}$, and obtain the configuration $C_{n+3} = C_{n+2} \setminus \{t'_{c_0}\} \cup \{t_s\}$, where $t_s$ matches $(unsafe?[x_1] \ldots unsafe?[x_k]c![\overline{x}], id_s, E_s)$ with $E_s(c) = E_{c_0}(c)$; we then use successively the threads $t'_{c_i}$ to obtain a configuration $C_{n+4} = (C_{n+2} \setminus \{t'_{c_i} \mid i \in [\![0;k]\!]\}) \cup \{t_m\}$, where $t_m$ matches $(x_0!^{s_k}[\overline{x}], id_m, E_m)$ with $E_m(x_i) = E''_{c_i}(c)$, for all $i \in [\![0;k]\!]$; by Prop. 2.2.1 it is the first time this syntactic component is tagged with the marker $id_m$; so we extend the definition of $\Phi_{\mathscr{M}}$ with $\Phi_{\mathscr{M}}(s_k, id_m) = id_!$); thus we have $\Pi^{\Phi_{\mathscr{M}}}_{\mathscr{M}}(N((i,s_k),id'_?,id_m)) = N((i,0),id_?,id_!)$; moreover the thread $t'_? = (y?^j[\overline{y}]P,id'_?,E'_?)$ is in $C_{n+4}$ with $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}} \circ E'_? = E_?$, and $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E''_{c_i}(c)) = (u_i,id_i)$, for each $i \in [\![1;k]\!]$; by Lem. C.1.1, there also exists $Cont'_? \in \beta(P,N((i,s_k),id'_?,id_m),E'_?[y_i \mapsto E''_{c_i}(c))])$ such that $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{C}}(Cont'_?) = Cont_?$; thus we perform the communication between the thread $t'_?$ and $t_m$ to obtain the configuration $C_{n+5} = C_{n+4} \setminus \{t_m\} \cup Cont'_?$.

- the computation sequence $C_n \longrightarrow^*_{\mathrm{e}} C_{n+4}$ does not involves thread of $\mathscr{S}_{\mathrm{I}}$, by Lem. C.1.2, we have $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{C}}(C_{n+4}) = A_{p-1}$; we conclude that $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{C}}(C_{n+5}) = A_p$; moreover $C_{n+4} \xrightarrow{i,j'}_{\mathrm{e}} C_{n+5}$ with $j' \notin \mathscr{L}_{\mathrm{I}}$; so $\Pi_\tau(\mathscr{S}_{\mathrm{I}},\Phi_{\mathscr{M}},\Phi_{\mathscr{N}})(C_0 \ldots C_{n+4} \xrightarrow{(i,j')}_{\mathrm{e}} C_{n+5}) = \tau'$.
- let $(u,id_u)$ be an unsafe name occurring in a state of $\tau$:
  - in the case that $(u,id_u) = (c_i,id_{c_i}$, for $i \in [\![0;k]\!]$, $t''_{c_i}$ is a thread of the state $C_{n+5}$ and matches $(\text{unsafe!}[c],id,E)$ where $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E(c)) = (u_i,id_i)$;
  - otherwise there is necessarily a thread in $C_n$ which matches $(\text{unsafe!}[y],id,E)$ with $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{N}}(E(y)) = (x,id_x)$ and this thread is still in $C_{n+5}$.

□

## C.2 Generalization

The following lemma establishes some soundness conditions:

**Lemma C.2.1.** *Let $\tau = C_0 \xrightarrow{\lambda_1}_{\mathrm{e}} \ldots \xrightarrow{\lambda_n}_{\mathrm{e}} C_n$ be a non standard computation sequence, with $C_0 \in \mathscr{C}^{\mathrm{e}}_0(\mathscr{S})$. We consider the following computation sequence:*

$$\Pi_\tau(\mathscr{S}_{\mathrm{I}},\Phi_{\mathscr{M}},\Phi_{\mathscr{N}})(\tau) = (A_0,U_0,F_0)\ldots(A_p,U_p,F_p).$$

*Then, $\forall (P,id,E) \in C_n$ with $lab(P) \notin \mathscr{L}_{\mathrm{I}}$:*

1. *$\forall x \in fn(P)$, we have $\Phi_{\mathscr{N}}(E(x)) \in U_n$;*
2. *$\Phi_{\mathscr{M}}(lab(P),id) \in F_n$.*

*Proof.* These two properties are easily proved by induction on $n$. □

We prove Thm. 6.3.14 that establishes the soundness of the generic context independent semantics.

**Theorem 6.3.14. (Soundness)** *Let $\tau = C_0 \dots C_n$ be a non standard computation sequence of a closed system that encloses the open system $\mathscr{S}_{\text{I}}$, with $C_0 \in \mathscr{C}_0$. Then $\Pi_\tau(\mathscr{S}_{\text{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(\tau) = (A_0, U_0, F_0) \dots (A_n, U_n, F_n)$ is a non standard computation sequence of the open system $\mathscr{S}_{\text{I}}$ and $(A_0, U_0, F_0) \in \mathscr{C}_0^{\text{o}}$.*

*Proof.* Soundness is ensured by construction: we prove Thm. 6.3.14 by induction on the length of the computation sequence:

1. We first prove that $(A_0, U_0, F_0) \in \mathscr{C}_0^{\text{o}}(\mathscr{S})$: we have $C_0 \in \mathscr{C}_0$; so there exists *continuation* $\in$ *init*$_s$ such that $C_0 = launch(continuation, \varepsilon, \emptyset)$. By definition of *launch*, we have $C_0 = \{(p, \varepsilon, [x \mapsto (E_s(x), \varepsilon)]) \mid (p, E_s) \in continuation\}$. Moreover, we have $A_0 = \Pi_{\text{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_0) = \{(p, \varepsilon, [x \mapsto \Pi_{\mathscr{L} \times \mathscr{M}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(()E_s(x), \varepsilon)]) \mid (p, E_s) \in continuation,\ p \in \mathscr{L}_{\text{p}}^{\text{I}}\}$. So, $A_0 = \Pi_{\text{C}}^{\Phi_{\mathscr{M}}, \Phi_{\mathscr{N}}}(C_0) = \{(p, \varepsilon, E(E_s)]) \mid (p, E_s) \in continuation,\ p \in \mathscr{L}_{\text{p}}^{\text{I}}\}$, where:

$$E(E_s) = \begin{cases} Dom(E_s) & \to & \mathscr{L} \times \mathscr{M} & \\ x & \mapsto & (E_s(x), \varepsilon) & \text{if } E_s(x) \in \mathscr{N}_{\text{I}} \\ x & \mapsto & \Phi_{\mathscr{N}}((E_s(x), \varepsilon)) & \text{otherwise.} \end{cases}$$

   So, since $\Phi_{\mathscr{N}} : (\mathscr{N} \times \mathscr{M} \to en)$, we obtain that there exists $E_0 \in \mathscr{V}_0 \to en$ such that

$$A_0 \in launch(\{(p, E_{s|Dom(E_s) \cap \mathscr{N}_{\text{I}}}) \in continuation \mid p \in \mathscr{L}_{\text{p}}^{\text{I}}\}, \varepsilon, E_0).$$

   Then, since $U_0 = en$ and $F_0 = \{t_n \mid n \in \mathbb{N}\}$, we obtain, by definition of $\mathscr{C}_0^{\text{o}}$, that $(A_0, U_0, F_0) \in \mathscr{C}_0^{\text{o}}$.

2. We now assume that Thm. 6.3.14 is satisfied for any non standard computation sequence $\tau$ containing at most $q$ computation steps, and we prove that it is also satisfied for any non standard computation sequence $\tau$ of containing $q+1$ computation steps: let $\tau = C_0 \dots C_q \xrightarrow{\lambda} C_{q+1}$ be a non standard computation sequence of length $q+1$; by the induction hypothesis $\Pi_\tau(\mathscr{S}_{\text{I}}, \Phi_{\mathscr{M}}, \Phi_{\mathscr{N}})(C_0 \dots C_q)$ is a non standard computation sequence of the open system $\mathscr{S}_{\text{I}}$, which we can denote:

$$(A_0, U_0, F_0) \dots (A_{q+1}, U_{q+1p}, F_{q+1}).$$

   We denote $\lambda = (\mathscr{R}, (t^k, pi^k, Ct_k), \tau^k)$, we denote $(t^k)_{1 \le k \le n} = (p^k, id^k, E^k)_{1 \le k \le n} \in C^q$ and $(pi^k)_{1 \le k \le n} = (s^k, (parameter_l^k)_l, (bd_l^k)_{k,l}, constraints^k, continutation^k)_{1 \le k \le n}$. We denote $\mathscr{R} = (n, components, compatibility, v\text{-}passing, broadcast)$.

   We have:

(a) $\forall k \in [\![1;n]\!],\ t^k \in C_q$;

(b) $\forall k \in [\![1;n]\!],\ exhibit(t^k, pi^k)$;

(c) $\forall k \in [\![1;n]\!],\ components(k) = s^k$;

(d) $sync((t^1,\dots,t^n),(parameter^k_l)_{k,l}, constraints)$ is satisfied.

Moreover, $C_{q+1} = subs(\tau, C_q \setminus removed\_threads \cup new\_threads)$ where:

- $\tau \in subs\_choice\left(\left(t^k\right)_k, \left(parameter^k_l\right)_{k,l}, broadcast\right)$;
- $removed\_threads = remove\left(\left(t^k, type(s^k)\right)_{1\le k\le n}\right)$;
- $new\_threads = \bigcup_{1\le k\le n} launch\left(Ct^k, \overline{id}^k, \overline{E}^k\right)$,
  with $\forall k \in [\![1;n]\!]$:
  - $Ct_k \in continuation^k$;
  - $\overline{id}^k = marker\left(type(s_k), \left(p^{k'}, id^{k'}, E^{k'}\right)_{1\le k'\le n}, k\right)$;
  - $\overline{E}^k$ is defined as:
    $vpassing(k, (t^{k'})_{1\le k'\le n}, (bd^k_l)_l, (parameter^{k'}_l)_{k',l}, communications)$.
- $\forall k \in [\![1;n;]\!],\ \alpha_k = (t^k, pi^k, Ct_k)$

We denote by $\mathscr{K}_s$ the set $\{i \in \mathbb{N} \mid 1 \le i \le n,\ t^i \in \mathscr{L}^{\mathrm{I}}_{\mathrm{p}}\}$ and by $\mathscr{K}_c = [\![1;n]\!] \setminus \mathscr{K}_s$. For any $k \in [\![1;n]\!]$, we define:

(a) $\bar{t}^k = \begin{cases} \Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(t^k) & \text{if } k \in \mathscr{K}_s \\ (0, \Phi_{\mathscr{M}}(p^k, id^k), [S_i \mapsto E^k(parameter^k_i)]) & \text{otherwise} \end{cases}$

(b) $\overline{pi}^k = \begin{cases} pi^k & \text{if } k \in \mathscr{K}_s \\ (s^k, (S_i), (bd^k_l)_{k,l}, \emptyset, \{\emptyset\}) & \text{otherwise} \end{cases}$;

(c) $\overline{ct}_k = \begin{cases} \emptyset & \text{if } k \in \mathscr{K}_s \\ Ct_k & \text{otherwise} \end{cases}$.

We have:

(a) for any $k \in \mathscr{K}_s$, $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(\bar{t}^k) \in A_q$ (since $p^k \in \mathscr{L}^{\mathrm{I}}_{\mathrm{p}}$), moreover, $exhibit(\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathrm{t}}(\bar{t}^k), pi^k)$ (since $\Pi^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}_{\mathscr{L}\times\mathscr{M}}$ is a bijection).

(b) for any $k, k' \in \mathscr{K}_c$, we have $\bar{t}^k \in\in \textsc{context\_thread}(U,F)$ (by Lem. C.2.1), we have $\textsc{context\_partial\_int}(\overline{pi}^k)$ (by Def.). we have $id_k = id'_k \implies k = k'$ by marker unambiguity, and since $\Phi_{\mathscr{M}}$ is a bijection.

(c) $\forall k \in [\![1;n]\!]$, $components(k) = s^k$;

(d) $sync((\bar{t}^1, \dots, \bar{t}^n), (parameter^k_l)_{k,l}, constraints)$ is satisfied.

So the computation step is enabled in the context independent semantics. Then we can apply projection over the sustitution $\tau$. This define a computation step in the context independent semantics such that: $(A_p, U_p, F_p) \overset{(\mathscr{R},(\bar{t}^k,\overline{pi}^k,\overline{ct}^k),\overline{tau})}{\looparrowright} (\Pi_{\mathrm{C}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(()C_{p+1}), U', F')$. The set $U'$ obtained by applying the substitution $\Pi_{\mathrm{SUBS}}^{\Phi_{\mathscr{M}},\Phi_{\mathscr{N}}}(\tau)$ over the union between $U$ and the set of values binds to variable of thread context during the interaction. Thus, $U' = U_{p+1}$, the union of $U$ and the channel spied by the system, by using communication. Moreover, by definition, we get that $F' = F_{p+1}$.

$\square$

# Appendix D

# Abstraction proves

## D.1 Environment abstraction

### D.1.1 Control flow analysis

#### D.1.1.1 Atom abstraction

**Theorem 8.2.1.** *These primitives satisfy the soundness assumptions of Sect. 8.1.1.1.*

*Proof.* Let $V \in \wp(\mathscr{V})$ be a finite set of variables.

- *initial environment abstraction:* We want to prove that $\{(\varepsilon, \emptyset)\} \subseteq \gamma_\emptyset(\varepsilon_\emptyset^\sharp)$.

  We have $\varepsilon_\emptyset^\sharp = (\varepsilon_1^{\mathscr{M}}, \emptyset)$. Then by soundness of $\varepsilon_1^{\mathscr{M}}$, we have $\varepsilon \in \gamma_1^{\mathscr{M}}(\varepsilon_1^{\mathscr{M}})$. Moreover, $\forall x \in \emptyset$, $\forall y \in \mathscr{L}$, $\emptyset(x) = (y, id_y) \implies (id, id_y) \in \gamma_2^{\mathscr{M}}(\emptyset(x,y))$. We conclude that $\{(\varepsilon, \emptyset)\} \subseteq \gamma_\emptyset(\varepsilon_\emptyset^\sharp)$.

- *abstract restriction:* Let $x$ be a variable in $\mathscr{V} \setminus V$, let $l$ be a label in $\mathscr{L}$ and $A = (id^\sharp, f)$ be an abstract element in $Atom_V^\sharp$, we want to prove that:

  $$\left\{(id,E) \in Env^{\mathscr{M}}_{V\cup\{x\}} \;\middle|\; \begin{array}{l}(id, E_{|V}) \in \gamma_V(A),\\ E(x) = (l, id)\end{array}\right\} \subseteq \gamma_{V\cup\{x\}}(\nu^\sharp(x,l,A)).$$

  We denote $(\overline{id}, \overline{f}) = \nu^\sharp(x,l,A)$. By definition of $\nu^\sharp$, we have $\overline{id} = id$ and $\overline{f}(y,m) = \begin{cases} f(y,m) & \text{if } y \neq x \\ \bot_2^{\mathscr{M}} & \text{if } x = y \text{ and } m \neq l. \\ \text{DIAG}(\text{PAIR}(a,a)) & \text{otherwise}\end{cases}$ Let $(id, E) \in Env^{\mathscr{M}}_{V\cup\{x\}}$ be a pair such that $(id, E_{|V}) \in \gamma_V(A)$ and $E(x) = (l, id)$. By definition of $\gamma_V$, we have $id \in \gamma_1^{\mathscr{M}}(id^\sharp)$ and for any $x \in V$, $E(x) = (y, id_y) \implies (id, id_y) \in$

$\gamma_2^{\mathscr{M}}(f(x,y))$. This way, we have $id \in \gamma_1^{\mathscr{M}}(\overline{id}^\sharp)$. Moreover, let $y \in V \cup \{x\}$ be a variable. We denote $E(y) = (m, id_y)$. We want to prove that $(id, id_y) \in \gamma_2^{\mathscr{M}}(\overline{f}(y,m))$.

1. in the case when $y \in V$, we have $E(y) = E_{|V}(y)$ and $\overline{f}(y,m) = f(y,m)$. By definition of $\gamma_V^{\mathscr{M}}$, we have $(id, id_y) \in f(y,m)$. so $(id, id_y) \in \gamma_2^{\mathscr{M}}(\overline{f}(y,m))$.
2. in the case when $y = x$, we have $E(y) = (l, id)$. By definition, we have $\overline{f}(y,m) = \text{DIAG}(\text{PAIR}(a,a))$. Then we have $id \in \gamma_1^{\mathscr{M}}(a)$. By definition of PAIR, we have $(id, id_y) \in \text{PAIR}(a,a)$. Thus since $id = id_y$, by definition of DIAG, we have $(id, id_y) \in \overline{f}(y,m)$.

We conclude that

$$\left\{ (id, E) \in Env^{\mathscr{M}}_{V \cup \{x\}} \;\middle|\; \begin{array}{l} (id, E_{|V}) \in \gamma_V(A), \\ E(x) = (l, id) \end{array} \right\} \subseteq \gamma_{V \cup \{x\}}(\nu^\sharp(x, l, A)).$$

- *abstract garbage collection:* Let $X$ be a finite subset of $\mathscr{V}$ and $A$ be an element of $Atom_V^\sharp$, the abstract projection, we want to prove that:

$$\{(id, E_{|V \cap X}) \in Env_X^{\mathscr{M}} \mid (id, E) \in \gamma_V(A)\} \subseteq \gamma_{V \cap X}(\text{GC}^\sharp(X, A)).$$

We denote $A = (id^\sharp, f)$. We have $\text{GC}^\sharp(X, A) = (id^\sharp, f_{|(V \cap X) \times \mathscr{L}})$. Let $(id, E) \in \gamma_V^{\mathscr{M}}(A)$ be a pair. We want to prove that $(id, E_{|V \cap X}) \in \gamma_{V \cap X}^{\mathscr{M}}(\text{GC}^\sharp(X, A))$. We have $id \in \gamma_1^{\mathscr{M}}(id^\sharp)$, by definition of $\gamma_V^{\mathscr{M}}$. Then for any $x \in V \cap X$, we denote $E_{|V \cap X}(x) = (l, id_x)$. We also have $E(x) = (l, id_x)$, so by definition of $\gamma_V^{\mathscr{M}}$ we conclude that $(id, id_x) \in \gamma_2^{\mathscr{M}}(f(x,l))$. Thus $(id, id_x) \in \gamma_2^{\mathscr{M}}(f_{(V \cap X) \times \mathscr{L}}(x, l)$.

□

### D.1.1.2 Molecule abstraction

**Theorem 8.2.2.** *These primitives satisfy the soundness assumptions of Sect. 8.1.2.*

*Proof.* We prove the soundness of each primitive as follows:

- *abstract injection:* Let $V$ be a finite set of variables. We want to prove that:

$$\forall A \in Atom_V^\sharp,\ \gamma_V(A) \subseteq \gamma_{(V)}(\text{INJ}^\sharp(A)).$$

Let $(id, E)$ be an element in $\gamma_V(A)$ We have $\text{INJ}^\sharp(A) = ([1 \rightarrow A], \emptyset, \emptyset, \emptyset, [\emptyset_{\mathscr{L}} \rightarrow \emptyset^\sharp_{\mathscr{M}}])$.

1. For any $i \in \{1\}$, we have $(id,E) \in \gamma_V([1 \rightarrow A](i))$.
2. We introduce the map $\sigma$ that interprets the variables as follows:

$$\sigma : \begin{cases} (I,1) \rightarrow (p,id) \\ (a,1) \rightarrow E(a). \end{cases}$$

   Then

   – for any $C \in \emptyset$, we have:

$$\forall x,y \in C,\ \sigma(x) = \sigma(y);$$

   – for any $(C_1,C_2) \in \emptyset$, we have:

$$\forall x \in C_1, y \in C_2,\ \sigma(x) \neq \sigma(y).$$

3. we have $\gamma_{\emptyset}^{\mathscr{M}}([\emptyset_{\mathscr{L}} \rightarrow \emptyset^{\sharp}_{\mathscr{M}}](\emptyset_{\mathscr{L}})) = \gamma_{\emptyset}^{\mathscr{M}}(\emptyset^{\sharp}_{\mathscr{M}}) = \top^{\mathscr{M}}$

- *abstract product:* Let $m$ and $n$ be two integers. Let $(U_i) \in (\wp(\mathscr{V}))^m$ and $(V_i) \in (\wp(\mathscr{V}))^n$ be two tuples of interfaces. We denote by $(W_i) \in (\wp(\mathscr{V}))^{m+n}$ the tuple of interfaces that is defined by $W_i = U_i$ when $1 \leq i \leq m$ and $W_{i+m} = V_i$ when $1 \leq i \leq n$. Let $A = (f_A, S_A, P_A, E_A, r_A)$ and $B = (f_B, S_B, P_B, E_B, r_B)$ be two abstract elements such that $A \in Molecule^{\sharp}_{(U_i)_{1\leq i\leq m}}$ and $B \in Molecule^{\sharp}_{(V_i)_{1\leq i\leq n}}$. We write $A \bullet B = (f_C, S_C, P_C, E_S, r_C)$. We want to prove that:

$$\left\{ (e_i)_{i\in[\![1;m+n]\!]} \;\middle|\; \begin{array}{l} (e_i)_{1\leq i\leq m} \in \gamma_{(U_i)}(A) \\ (e_{i+m})_{1\leq i\leq n} \in \gamma_{(V_i)}(B) \end{array} \right\} \subseteq \gamma_{(W_i)}(A \bullet B).$$

  Let $(e_i)_{i\in[\![1;m+n]\!]}$ such that:

$$\begin{array}{l} (e_i)_{1\leq i\leq m} \in \gamma_{(U_i)}(A) \\ (e_{i+m})_{1\leq i\leq n} \in \gamma_{(V_i)}(B). \end{array}$$

  For each $i \in [\![1;m+n]\!]$, we write $e_i = (id_i, E_i)$.

  1. Let $k \in [\![1;m+n]\!]$ be an integer.

     – If $k \leq m$ then we have $(e_i)_{1\leq i\leq m} \in \gamma_{(U_i)}(A)$; so $e_k \in \gamma_{U_k}(f_A(k))$; since both $f_A(k) = f_C(k)$ and $U_k = W_k$, we get that $e_k \in \gamma_{W_k} f_C(k)$.

     – If $k > m$ then we have $(e_{i+m})_{1\leq i\leq n} \in \gamma_{(V_i)}(A)$; so $e_k \in \gamma_{V_{k-m}}(f_B(k-m))$; since both $f_B(k-m) = f_C(k)$ and $V_{k-m} = W_k$, we get that $e_k \in \gamma_{W_k} f_C(k)$.

So $e_k \in \gamma_{W_k} f_C(k)$.

2. We introduce the map $\sigma_A$ that interprets the variables as follows:

$$\sigma_A : \begin{cases} (I,k) \to (p_k, id_k) \\ (a,k) \to E_k(a). \end{cases}$$

We introduce the map $\sigma_B$ that interprets the variables as follows:

$$\sigma_B : \begin{cases} (I,k) \to (p_{k+m}, id_{k+m}) \\ (a,k) \to E_{k+m}(a). \end{cases}$$

We introduce the map $\sigma_C$ that interprets the variables as follows:

$$\sigma_C : \begin{cases} (I,k) \to (p_k, id_k) \\ (a,k) \to E_k(a). \end{cases}$$

We introduce $\sigma^{B\to C} \in B \to \{(x,k+m) \mid x \in V_k \cup \{I\}\}$ and $\sigma^{C\to B} \in \{(x,k+m) \mid x \in V_k \cup \{I\}\} \to B$ as follows:

$$\sigma^{B\to C}((x,k)) = (x,k+n),\ \forall x \in V_k \cup \{I\};$$
$$\sigma^{C\to B}((x,k+n)) = (x,k),\ \forall x \in V_k \cup \{I\}.$$

Then:

(a) Let $C \in P_C$ be an equivalence class. Let $x,y$ be two variables in $C$. If $C \in P_A$, then we have $\sigma_C(x) = \sigma_A(x) = \sigma_A(y) = \sigma_C(y)$, since $(p_k, id_k, E_k)_{1\leq k\leq m} \in \gamma_{(U_i)}(A)$. Otherwise, there exists $C' \in P_B$ such that $C = \{(x,k+m) \mid (x,k) \in C'\}$. Moreover, we denote $x = (x', k_x + m)$ and $y = (y', k_y + m)$. We have $\sigma_C(x) = \sigma_B(x', k_x) = \sigma_B(y', k_y) = \sigma_C(y)$, since $(id_{k+m}, E_{k+m})_{1\leq k\leq n} \in \gamma_{(V_i)}(B)$.

(b) Let $(C_1, C_2) \in P_C$ be a disequality constraint. Let $x \in C_1$ and $y \in C_2$ be two variables. If $(C_1, C_2) \in E_A$, then we have $\sigma_C(x) = \sigma_A(x) \neq \sigma_A(y) = \sigma_C(y)$, since $(p_k, id_k, E_k)_{1\leq k\leq m} \in \gamma_{(U_i)}(A)$. Otherwise, there exists $(C_1', C_2') \in E_B$ such that $C_1 = \{(x,k+m) \mid (x,k) \in C_1'\}$ and $C_2 = \{(x,k+m) \mid (x,k) \in C_2'\}$. Moreover, we denote $x = (x', k_x + m)$ and $y = (y', k_y + m)$. We have $\sigma_C(x) = \sigma_B(x', k_x) \neq \sigma_B(y', k_y) = \sigma_C(y)$, since $(id_{k+m}, E_{k+m})_{1\leq k\leq n} \in \gamma_{(V_i)}(B)$.

3. Let us introduce the function $t \in P_C \to \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(a,k) \in C$ and a marker $id$ such that $E_k(a) = (l, id)$ or by $t(C) = p_k$ when there exists $(I,k) \in C$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P \to \mathscr{M}$ is defined by

$f_{\mathscr{M}}(C) = id_C$ when there exists $(a,k) \in C$ such that $E_k(a) = (l_C, id_C)$ or when there exists $(I,k) \in C$ such that $id_k = id_C$. Then we have $f_{\mathscr{M}|P_A} \in \gamma^{\mathscr{M}}_{P_A}(r_A(t_{|P_A}))$, we have $f_{\mathscr{M}|\{(x,k+m) \mid 1 \leq k \leq n,\, x \in V_k \cup \{I\}\}} \circ \sigma^{B \to C} \in \gamma^{\mathscr{M}}_{P_B}([r_B(\sigma^{C \to B} t_{|\{(x,k+m) \mid 1 \leq k \leq n,\, x \in V_k \cup \{I\}\}})])$. By definition of the abstract quotient, we have: $\text{QUOTIENT}(\sigma^{B \to C}, f_{\mathscr{M}|\{(x,k+m) \mid 1 \leq k \leq n,\, x \in V_k \cup \{I\}\}}) \in \gamma^{\mathscr{M}}_{\overline{P_B}}([r_B(\sigma^{C \to B} t_{|\{(x,k+m) \mid 1 \leq k \leq n,\, x \in V_k \cup \{I\}\}})])$. Then since, $P_A \cap \overline{P_B} = \emptyset$, we have: $f_{\mathscr{M}|P_C} \in \gamma^{\mathscr{M}}_{P_C}(r_A(t_{|P_A}) \otimes \text{QUOTIENT}(\sigma^{B \to C}, f_{\mathscr{M}|\{(x,k+m) \mid 1 \leq k \leq n,\, x \in V_k \cup \{I\}\}}))$. Thus $f_{\mathscr{M}|P_C} \in \gamma^{\mathscr{M}}_{P_C}(r_C(t))$.

- *abstract projections:* the primitive $\text{PROJ}^\sharp$ extracts the description of a thread from the description of a tuple of threads. Let $(V_i) \in (\wp(\mathscr{V}))^n$ be an $n$-tuple of interfaces. Let $A \in Molecule^\sharp_{(V_i)_{1 \leq i \leq n}}$ be an abstract element. Let $k$ be an integer such that $k \leq n$. We want to prove that:

$$\left\{ (id_k, E_k) \;\middle|\; \exists (id_i, E_i)_i \in \gamma_{(V_i)_i}(A) \right\} \subseteq \gamma_{V_k}(\text{PROJ}^\sharp(k,A));$$

  We denote $A = (f,S,P,E,r)$, $f(k) = (a_0,b_0)$ and $\text{PROJ}^\sharp(k,A) = (a,b)$. For any $X \in S$, we denote by $C(X)$ the unique class in $P$ such that $X \in C(X)$. Let $(id_i,E_i)_i \in \gamma_{(V_i)_i}(A)$, we want to prove that: $(id_k,E_k) \in \gamma_{V_k}(\text{PROJ}^\sharp(k,A))$:

  1. Since $(id_i,E_i)_i \in \gamma_{(V_i)_i}(A)$, we have $(id_k,E_k) \in \gamma_{V_k}(f(k))$. So $(id_k,E_k) \in \gamma^{\mathscr{M}}_1(a_0)$.
     - In the case when $(I,k) \notin S$: we have $a = a_0$, so $id_k \in \gamma^{\mathscr{M}}_1(a)$.
     - Otherwise, we have $(id_i,E_i)_i \in \gamma_{(V_i)_i}(A)$. Let us introduce the function $t \in P \to \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(x,l) \in C$ and a marker $id$ such that $E_l(x) = (l,id)$ or by $t(C) = p_l$ when there exists $(I,l) \in C$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P \to \mathscr{M}$ is defined by $f_{\mathscr{M}}(C) = id_C$ when there exists $(a,k) \in C$ such that $E_k(a) = (l_C, id_C)$ or when there exists $(I,k) \in C$ such that $id_k = id_C$. Since, $(id_i,E_i)_i \in \gamma_{(V_i)_i}(A)$, we have $f_{\mathscr{M}} \in \gamma^{\mathscr{M}}_P(r(t))$. We have $id_k = f_{\mathscr{M}}(C(I,k))$. By soundness of the primitive EXTRACT-SG, we have $f_{\mathscr{M}}(C(I,k)) \in \text{EXTRACT-SG}_{C((I,k))}(r(t))$. Thus $id_k \in \text{EXTRACT-SG}_{C((I,k))}(r(t))$. We can conclude that:

$$id_k \in \gamma^{\mathscr{M}}_1\left( \sqcup^{\mathscr{M}}_1 \{ \text{EXTRACT-SG}_{C((I,k))}(r(\sigma)) \mid \sigma \in P \to \mathscr{L} \} \right).$$

     Since $id_k \in \gamma^{\mathscr{M}}_1(a_0)$, we conclude by soundness of the abstract intersection that $id_k \in \gamma^{\mathscr{M}}_1(a)$.

2. Let $x \in V_k$, we denote $(y, id_x) = E_k(x)$. We want to prove that $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b(x,y))$.
   (a) In the case when $(I,k) \in S$ and $(x,k) \in S$:
   - We have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b_0(x,y))$, since $(id_k, E_k) \in \gamma_{V_k}^{\mathscr{M}}(f(k))$;
   - Let us introduce the function $t \in P \rightarrow \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(x,l) \in C$ and a marker $id$ such that $E_l(x) = (l, id)$ or by $t(C) = p_l$ when there exists $(I,l) \in C$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P \rightarrow \mathscr{M}$ is defined by $f_{\mathscr{M}}(C) = id_C$ when there exists $(a,k) \in C$ such that $E_k(a) = (l_C, id_C)$ or when there exists $(I,k) \in C$ such that $id_k = id_C$. We have $t(C((x,k)) = y$. Since, $(id_i, E_i)_i \in \gamma_{(V_i)_i}(A)$, we have $f_{\mathscr{M}} \in \gamma_P^{\mathscr{M}}(r(t))$. We have $id_k = f_{\mathscr{M}}(C(I,k))$ and $id_x = f\mathscr{M}(C(x,k))$. By soundness of the primitive EXTRACT-PR, we have $(f_{\mathscr{M}}(C((I,k))), f_{\mathscr{M}}(C((x,k)))) \in \text{EXTRACT-PR}_{C((I,k)),C((x,k))}(r(t))$. Thus $(id_k, id_x) \in \text{EXTRACT-PR}_{C((I,k)),C((x,k))}(r(t))$. We can conclude that the pair $(id_k, id_x)$ is an element of the set $\gamma_2^{\mathscr{M}}(A)$, where $A$ is defined as follows:
   $$\sqcup_2^{\mathscr{M}} \left\{ \text{EXTRACT-PR}_{C((I,k)),C((x,k))}(r(\sigma)) \;\middle|\; \begin{array}{l} \sigma \in P \rightarrow \mathscr{L} \\ \sigma(C((x,k))) = y \end{array} \right\}.$$

   By soundness of the abstract intersection, we have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}\left(\sqcap_2^{\mathscr{M}}\{b_0(x,y); A\}\right)$, where $A = \sqcup_2^{\mathscr{M}} \left\{ \text{EXTRACT-PR}_{C((I,k)),C((x,k))}(r(\sigma)) \;\middle|\; \begin{array}{l} \sigma \in P \rightarrow \mathscr{L} \\ \sigma(C((x,k))) = y \end{array} \right\}$.
   We conclude that $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b(x,y))$.

   (b) In the case when $(I,k) \in S$ and $(x,k) \notin S$:
   - We have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b_0(x,y))$, since $(id_k, E_k) \in \gamma_{V_k}^{\mathscr{M}}(f(k))$;
   - We have $id_k \in \gamma_1^{\mathscr{M}}(a)$ and $id_x \in \top_1^{\mathscr{M}}$. Thus, $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(\text{PAIR}(a, \top_1^{\mathscr{M}}))$.

   Since $b(x,y) = \sqcap_2^{\mathscr{M}}\{\text{PAIR}(a, \top_1^{\mathscr{M}}); b_0(x,y)\}$, we conclude that $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b(x,y))$.

   (c) In the case when $(I,k) \notin S$ and $(x,k) \in S$:
   - We have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b_0(x,y))$, since $(id_k, E_k) \in \gamma_{V_k}^{\mathscr{M}}(f(k))$;
   - Let us introduce the function $t \in P \rightarrow \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(x,l) \in C$ and a marker $id$

such that $E_l(x) = (l, id)$ or by $t(C) = p_l$ when there exists $(I, l) \in C$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P \to \mathscr{M}$ is defined by $f_{\mathscr{M}}(C) = id_C$ when there exists $(a, k) \in C$ such that $E_k(a) = (l_C, id_C)$ or when there exists $(I, k) \in C$ such that $id_k = id_C$. We have $t(C((x,k)) = y$. Since, $(id_i, E_i)_i \in \gamma_{(V_i)_i}(A)$, we have $f_{\mathscr{M}} \in \gamma_P^{\mathscr{M}}(r(t))$. We have $id_x = f_{\mathscr{M}}(C(x,k))$. By soundness of the primitive EXTRACT-SG, we have $f_{\mathscr{M}}(C((x,k))) \in \text{EXTRACT-SG}_{C((x,k))}(r(t))$. Thus $id_x \in \text{EXTRACT-SG}_{C((x,k))}(r(t))$. We can conclude that the element $id_x$ belongs to the set $\gamma_1^{\mathscr{M}}(A)$, where $A$ is defined as follows:

$$\sqcup_1^{\mathscr{M}} \left\{ \text{EXTRACT-SG}_{C((x,k))}(r(\sigma)) \;\middle|\; \begin{array}{l} \sigma \in P \to \mathscr{L} \\ \sigma(C((x,k))) = y \end{array} \right\}.$$

Moreover, we have $id_k \in \gamma_1^{\mathscr{M}}(a)$. By soundness of the abstract pair, we conclude that the pair $id_k, id_x)$ is an element of the set $\gamma_2^{\mathscr{M}}(A)$, where $A$ is defined as follows:

$$\text{PAIR}\left(a, \sqcup_1^{\mathscr{M}} \left\{ \text{EXTRACT-SG}_{C((x,k))}(r(\sigma)) \;\middle|\; \begin{array}{l} \sigma \in P \to \mathscr{L} \\ \sigma(C((x,k))) = y \end{array} \right\}\right)$$

By soundness of the abstract intersection, we have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}\left(\sqcap_2^{\mathscr{M}}\{b_0(x,y); A\}\right)$. We conclude that $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b(x,y))$.

(d) In the case when $(I, k) \notin S$ and $(x, k) \notin S$:

- We have $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b_0(x,y))$, since $(id_k, E_k) \in \gamma_{V_k}^{\mathscr{M}}(f(k))$;
- We have $id_k \in \gamma_1^{\mathscr{M}}(\top_1^{\mathscr{M}})$ and $id_x \in \top_1^{\mathscr{M}}$. Thus, $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(\text{PAIR}(\top_1^{\mathscr{M}}, \top_1^{\mathscr{M}}))$.

Since $b(x,y) = \sqcap_2^{\mathscr{M}}\{\text{PAIR}(\top_1^{\mathscr{M}}, \top_1^{\mathscr{M}}); b_0(x,y)\}$, we conclude that $(id_k, id_x) \in \gamma_2^{\mathscr{M}}(b(x,y))$.

- *abstract extension:* Let $(V_i)_{1 \le i \le n}$ be an $n$-tuple of interfaces, let $X$ be a subset of $\mathscr{V} \times [\![1;n]\!]$ and let $A$ be an abstract element in $Molecule^{\sharp}_{(V_i)}$. For each $i \in [\![1;n]\!]$, we define the set $U_i \subseteq \mathscr{V}$ of variables as $V_i \setminus \{x \mid (x,i) \in X\}$ and the set $W_i \subseteq \mathscr{V}$ of variables as $V_i \cup \{x \mid (x,i) \in X\}$. For each $i$ such that $1 \le i \le n$, the set $U_i$ is the set of the variables of the $i$-th interface that are kept unchanged whereas the set $W_i$ is the set of the variables of the updated $i$-th interface. We want to prove that:

$$\left\{ (id_i, E_i) \in \Pi(Env_{W_i}^{\mathscr{M}}) \;\middle|\; \begin{array}{l} \exists (id_i, E_i') \in \gamma_{(V_i)}(A), \\ \forall i \in [\![1;n]\!], \forall x \in U_i, \\ E_i'(x) = E_i(x) \end{array} \right\} \subseteq \gamma_{(W_i)}(\text{NEW}^{\sharp}_{\top}(X, A)).$$

Let $(id_i, E_i) \in \Pi(Env^{\mathscr{M}}_{W_i})$, such that:

$$\exists (id_i, \overline{E}_i) \in \gamma_{(V_i)}(A), \forall i \in [\![1;n]\!], \forall x \in U_i, E'_i(x) = E_i(x)$$

We denote $A = (f, S, P, E, r)$ and $\text{NEW}^{\sharp}_{\top}(X, A) = (f', S', P', E', r')$.

1. Let $k \in [\![1;n]\!]$ be an integer.
   we have $(id_i, \overline{E}_i)_{1 \leq i \leq m} \in \gamma_{(V_i)}(A)$; so $(id_k, \overline{E}_k) \in \gamma_{V_k}(f(k))$; then $id_k \in \gamma^{\mathscr{M}}_1(fst(f(k)))$, and for any $x \in W_i$, if $x \in U_i$, then $E_k(x) = \overline{E}_k(x)$, so $(id_k, snd(E_k(x))) \in \gamma^{\mathscr{M}}_2(snd(f_A(k)(x, fst(E_k(x)))))$, then, $(id_k, snd(E_k(x))) \in \gamma^{\mathscr{M}}_2(snd(f'(k)(x, fst(\overline{E}_k(x)))))$; otherwise, $snd(f'(k)(x, fst(\overline{E}_k(x))) = \text{PAIR}(fst(f(k)), \top^{\mathscr{M}}_1)$, so $(id_k, snd(E_k(x))) \in \gamma^{\mathscr{M}}_2(snd(f'(k)(x, fst(\overline{E}_k(x)))))$.
   So $f'_k \in \gamma_{W_k} f(k)$.
2. We introduce the map $\sigma$ that interprets the variables as follows:
   $$\sigma : \begin{cases} (I, k) \rightarrow (p_k, id_k) \\ (a, k) \rightarrow \overline{E}_k(a). \end{cases}$$
   We introduce the map $\sigma'$ that interprets the variables as follows:
   $$\sigma' : \begin{cases} (I, k) \rightarrow (p_k, id_k) \\ (a, k) \rightarrow E_k(a). \end{cases}$$
3. for any $C \in P'$, for any $(x, k), (y, l) \in C$, we have $x \in U_k \cup \{I`\}$, $y \in U_l \cup \{I\})$, and there exists $C' \in P$ such that $(x, k), (y, l) \in C'$, so $\sigma'(x) = \sigma'(y)$; then, $\sigma(x) = \sigma'(x)$ and $\sigma'(y) = \sigma(y)$, so $\sigma(x) = \sigma(y)$.
4. for any $(C_1, C_2) \in E$, for any $((x, k), (y, l)) \in (C_1, C_2)$, we have $x \in U_k$, $y \in U_l)$, and there exists $C'_1, C'_2 \in E'$ such that $((x, k), (y, l)) \in E'$, so $\sigma'(x) \neq \sigma'(y)$; then, $\sigma(x) = \sigma'(x)$ and $\sigma(y) = \sigma'(y)$, so $\sigma(x) \neq \sigma'(y)$.
5. Let us introduce the function $t \in P' \rightarrow \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(a, k) \in C$ and a marker $id$ such that $E_k(a) = (l, id)$ or by $t(C) = p_k$ when there exists $(I, k) \in C$. The same way, let us introduce denote by $f_{\mathscr{M}} \in P' \rightarrow \mathscr{M}$ is defined by $f_{\mathscr{M}}(C) = id_C$ when there exists $(a, k) \in C$ such that $E_k(a) = (l_C, id_C)$ or when there exists $(I, k) \in C$ such that $id_k = id_C$. We define $\bar{t} \in P \rightarrow \mathscr{L}$ that is defined by $t(C) = l$ when there exists $(a, k) \in C$ and a marker $id$ such that $\overline{E}_k(a) = (l, id)$ or by $t(C) = p_k$ when there exists $(I, k) \in C$. We have for any $C \in P$ such that $C \not\subseteq X$, $t(C \setminus X) = \bar{t}(C)$. Thus, $\gamma^{\mathscr{M}}_{P'}(\psi(r(\bar{t}))) \subseteq \gamma^{\mathscr{M}}_{P'}(r'(\bar{t}))$, where $\psi(a)$ is defined as:
   $$\text{QUOTIENT}([C \mapsto C \setminus X], \text{PROJ}(\{C \in P \mid C \not\subseteq X\}, a)).$$

Then $f_{\mathscr{M}} \in \gamma_{P'}^{\mathscr{M}}(\psi(r(\bar{t})))$, so $f_{\mathscr{M}} \in \gamma_{P'}^{\mathscr{M}}(r'(\bar{t}))$.

$\square$

## D.2 Occurrence counting abstraction

In this section we give the proof of Thm. 9.2.4 which establishes the soundness of our occurrence counting analysis.

**Theorem 9.2.4.** *$(\mathscr{N}_{\mathscr{V}_c}, \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}, \sqcup_{\mathscr{N}_{\mathscr{V}_c}}, \bot_{\mathscr{N}_{\mathscr{V}_c}}, \gamma_{\mathbb{N}^{\mathscr{V}_c}} \circ \gamma_{\mathscr{N}_{\mathscr{V}_c}}, C_0^{\mathscr{N}_{\mathscr{V}_c}}, \rightsquigarrow_{\mathscr{N}_{\mathscr{V}_c}}, \nabla_{\mathscr{N}_{\mathscr{V}_c}})$ is an abstraction.*

*Proof.* The tuple $(\mathscr{N}_{\mathscr{V}_c}, \sqsubseteq_{\mathscr{N}_{\mathscr{V}_c}}, \sqcup_{\mathscr{N}_{\mathscr{V}_c}}, \bot_{\mathscr{N}_{\mathscr{V}_c}}, \gamma_{\mathbb{N}^{\mathscr{V}_c}} \circ \gamma_{\mathscr{N}_{\mathscr{V}_c}}, C_0^{\mathscr{N}_{\mathscr{V}_c}}, \rightsquigarrow_{\mathscr{N}_{\mathscr{V}_c}}, \nabla_{\mathscr{N}_{\mathscr{V}_c}})$ satisfies Def. 7.2.1:

- Props. (1)(2)(3)(7) are satisfied by our assumptions;
- Prop. (4) is satisfied since both $\gamma_{\mathbb{N}^{\mathscr{V}_c}}$ and $\gamma_{\mathscr{N}_{\mathscr{V}_c}}$ are monotonic;
- Prop. (5) is satisfied:
- Prop. (6) is satisfied: let $v^\sharp$ be an abstract configuration, $(u,C)$ be in the concretization $(\gamma_{\mathbb{N}^{\mathscr{V}_c}} \circ \gamma_{\mathscr{N}_{\mathscr{V}_c}})(v^\sharp)$, $\lambda$ be a transition label and $\overline{C}$ be another configuration such that $C \overset{\lambda}{\looparrowright} \overline{C}$, we must construct $\bar{v}^\sharp$ such that $(u.\lambda, \overline{C}) \in \gamma(\bar{v}^\sharp)$ and $v^\sharp \overset{\lambda}{\rightsquigarrow}_{\mathscr{N}_{\mathscr{V}_c}} \bar{v}^\sharp$.

  Let $(\mathscr{R}, ((p^k, id^k, E^k), pi^k, Ct_k)_{1 \le k \le n})$ be the pair such that $\lambda = (\mathscr{R}, ((p^k, id^k, E^k), pi^k, Ct_k)_{1 \le k \le n})$.

  Let $(pi^k) = (s^k, (parameter_l^k)_l, (bd_l^k)_{k,l}, constraints^k, continuation^k)_{1 \le k \le n}$ be a sequence of partial interactions. We have $C \overset{\lambda}{\looparrowright} \overline{C}$. So according to Fig. 4.1 on page 62 and Def. 4.4.1 on page 57, we have $pi^k \in interaction(p^k)$ and $(p^k, id^k, E^k) \in C$ for any $k \in [\![1;n]\!]$. According to Fig. 4.1, there exists $\tau \in subs\_choice\left((t^k)_k, (parameter_l^k)_{k,l}, broadcast\right)$, such that:

  $$\overline{C} = subs(\tau, C \setminus rem\_threads \cup new\_threads)$$

  where:

  - $subs(\tau, C) = \{(q, id, \tau \circ E) \mid (q, id, E) \in C\}$;
  - $rem\_threads = remove\left((t^k, type(s^k))_{1 \le k \le n}\right)$;

- $new\_threads = \bigcup_{1\leq k\leq n} launch\left(Ct^k, \overline{id}^k, \overline{E}^k\right)$,
  with $\forall k \in [\![1;n]\!]$:
  - $\overline{id}^k = marker\left(type(s_k), \left(p^{k'}, id^{k'}, E^{k'}\right)_{1\leq k'\leq n}, k\right)$;
  - the element $\overline{E}^k$ is defined as follows:

    $vpassing(k, (t^{k'})_{1\leq k'\leq n}, (bd_l^k)_l, (parameter^{k'}_l)_{k',l}, communications)$.

We introduce the configuration $D = C \setminus rem\_threads \cup new\_threads$.

We have $(u,C) \in (\gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}})(v^\sharp)$; so $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u,C) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(v^\sharp)$; since $(p^k, id^k, E^k) \in C$ is a sequence of distinct threads, we have $\forall k \in [\![1;n]\!]$, $Card\{l \mid p^k = p^l\} \leq \Pi_{\mathbb{N}^{\mathcal{V}_c}}(u,C)_{p^k}$; so $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u,C) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}((p^k)_{1\leq k\leq n}, v^\sharp))$; since $\gamma_{\mathcal{N}_{\mathcal{V}_c}}$ is strict, we can conclude that $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}((p^k)_{1\leq k\leq n}, v^\sharp) \neq \bot_{\mathcal{N}_{\mathcal{V}_c}}$;

Now, we define $\overline{v}^\sharp$ as:

$$\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, v^\sharp) +^\sharp Transition +^\sharp Launched -^\sharp Consumed,$$

where:

- $Transition = 1_{\mathcal{N}_{\mathcal{V}_c}}(\psi(\lambda))$;
- $Launched = \Sigma^\sharp\left(\beta_{\mathcal{N}_{\mathcal{V}_c}}((continuation^k)_k))_{1\leq k\leq n})\right)$;
- $Consumed = \Sigma^\sharp(1_{\mathcal{N}_{\mathcal{V}_c}}(p_k))_{k\in\{k' \mid 1\leq k'\leq n,\, type(s_{k'})\neq replication\}}$.

We know that $v^\sharp \overset{\alpha_\lambda(\lambda)}{\rightsquigarrow}_{\mathcal{N}_{\mathcal{V}_c}} v^\sharp$. We now prove that $(u.\lambda, D) \in (\gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}})(\overline{v}^\sharp)$: we have $D = C \setminus rem\_threads \cup new\_threads$; thus, the tuple $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u.\lambda, D)$ is equal to:

$$\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u,C) -^\sharp \Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, rem\_threads) +^\sharp \Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, new\_threads) +^\sharp \Pi_{\mathbb{N}^{\mathcal{V}_c}}(\lambda, \emptyset);$$

moreover, we have:

1. $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u,C) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, v^\sharp))$, since $(u,C) \in (\gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}})(\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, v^\sharp))$;
2. $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, rem\_threads) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(consumed)$ (by using both Prop.9.2.2 on page 250 and marker unambiguity);
3. $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\lambda, \emptyset) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(Transition)$;

4. $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, new\_threads) = \sum^\sharp\left((\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, Ct^k))_{1 \leq k \leq n}\right)$ (by using both Prop. 9.2.2 and marker unambiguity); then, $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, Ct^k) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\chi^\sharp(Ct^k))$. Since $Ct_k \in continuation^k$ and by soundness of the abstract union and of the abstract characteristic function, we have also: $\gamma_{\mathcal{N}_{\mathcal{V}_c}}(\chi^\sharp(Ct_k)) \sqsubseteq_{\mathcal{N}_{\mathcal{V}_c}} \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\beta_{\mathcal{N}_{\mathcal{V}_c}}(continuation^k))$. Thus, $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, Ct^k) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\beta_{\mathcal{N}_{\mathcal{V}_c}}(continuation^k)))$. By Prop. 9.2.2 and marker unambiguity, we get that: $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\varepsilon, new\_threads) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\beta_{\mathcal{N}_{\mathcal{V}_c}}(continuation^k))$;

then, by Prop. 9.2.2 and marker unambiguity, we get that:

$$\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u.\lambda, C \setminus rem\_threads \cup new\_threads) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\overline{v}^\sharp);$$

since, global substitution does not change neither occurrence number of threads, nor their program point, we can conclude that $\Pi_{\mathbb{N}^{\mathcal{V}_c}}(u.\lambda, \overline{C}) \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\overline{v}^\sharp)$; thus,

$$(u.\lambda, \overline{C}) \in (\gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}})(\overline{v}^\sharp).$$

□

# Bibliographie

[1] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Proc. POPL'01*. ACM Press, 2001.

[2] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols : The spi calculus. In *Proc. CCS'97*. ACM Press, 1997.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols : The spi calculus. *Information and Computation*, 148(1), 1999.

[4] Benjamin Y. Y. Aziz. A static analysis of cryptographic processes : the denotational approach. *Journal of Logic and Algebraic Programming, special issue on Modelling and Verification of Cryptographic Protocols*, ? ?( ? ?), ? ? to appear.

[5] Benjamin Y. Y. Aziz. *A Static Analysis Framework for Security Properties in Mobile and Cryptographic Systems*. PhD thesis, Dublin City University, 2003.

[6] Benjamin Y. Y. Aziz and Geoffrey W. Hamilton. A privacy analysis for the pi-calculus : The denotational approach. In *Proc. SAVE'02*, number 94 in Datalogiske Skrifter. Department of Computer Science, Roskilde University, 2002.

[7] Bruno Blanchet. From Secrecy to Authenticity in Security Protocols. In *Proc. SAS'02*, volume 2477 of *Lecture Notes on Computer Science*. Springer Verlag, 2002.

[8] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proc. PLDI'03*. ACM Press, 2003.

[9] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the $\pi$-calculus. In *Proc. CONCUR'98*, LNCS. Springer-Verlag, 1998.

[10] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis of processes for no read-up and no write-down. In *Proc. FoSSaCS'99*, LNCS. Springer-Verlag, 1999.

[11] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for the $\pi$-calculus with applications to security. *Information and Computation*, 165, 2000.

[12] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Boxed ambients. In *In Proc. TACS'01*, number 2215 in LNCS, pages 38–63. Springer, 2001.

[13] Michele Bugliesi, Giuseppe Castagna, and Silvia Crafa. Access control for mobile agents : The calculus of boxed ambients. *ACM Transactions on Programming Languages and Systems*, 26(1), 2004.

[14] Luca Cardelli. Brane calculi. In *Proc. BIO-CONCUR'03*, ENTCS. Elsevier Science Publishers. to appear.

[15] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Ambient groups and mobility types. In *Proc. TCS'00*, LNCS. Springer-Verlag, 2000.

[16] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *Proc. CONCUR'00*, LNCS. Springer-Verlag, 2000.

[17] Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 1998.

[18] Sagar Chaki, Sriram K. Rajamani, and Jakob Rehof. Types as models : Model checking message-passing programs. In *Proc. POPL'02*. ACM, 2002.

[19] Witold Charatonik, Andrew D. Gordon, and Jean Marc Talbot. Finite-control mobile ambients. In *Proc. ESOP'02*, number 2305 in LNCS. Springer-Verlag, 2002.

[20] Chong-Kan Chiu and Jimmy Ho-Man Lee. Interval linear constraint solving using the preconditioned interval Gauss-Seidel method. In *Proc. ICLP'95*, Logic Programming. The MIT Press, 1995.

[21] Patrick Cousot. Semantic foundations of program analysis. In *Program Flow Analysis : Theory and Applications*, chapter 10. Prentice-Hall, Inc., 1981.

[22] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130. Dunod, Paris, France, 1976.

[23] Patrick Cousot and Radhia Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. POPL'77*. ACM Press, 1977.

[24] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4), 1992.

[25] Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening-narrowing approaches to abstract interpretation. In *Proc. PLILP'92*, LNCS. Springer-Verlag, 1992.

[26] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proc. POPL'78*. ACM Press, 1978.

[27] Mads Dam. Model checking mobile processes. *Inf. Comput.*, 129(1) :35–51, 1996.

[28] Vincent Danos and Sylvain Pradalier. Projective brane calculus. In *Proc. CMSB'04*, Lecture Notes on Computer Science. Springer Verlag. to appear.

[29] Nicolaas Govert de Bruijn. Lambda-calculus notation with nameless dummies : a tool for automatic formula manipulation with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34(5), 1972.

[30] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B : Formal Models and Sematics (B)*. Elsevier Science Publishers, 1990.

[31] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols (extended abstract). In *Proc. FOCS'81*. IEEE Press, 1981.

[32] Jérôme Feret. Abstract interpretation of mobile systems. *Journal of Logic and Algebraic Programming, special issue on pi-calculus*, ? ?( ? ?), ? ? to appear.

[33] Jérôme Feret. *amb*-s.a : a static analyzer for mobile ambients. Prototype, `http://www.di.ens.fr/~feret/prototypes/`.

[34] Jérôme Feret. $\pi$-s.a : a static analyzer for the $\pi$ calculus. Prototype, `http://www.di.ens.fr/~feret/prototypes/`.

[35] Jérôme Feret. Conception de $\pi$-*sa* : un analyseur statique générique pour le $\pi$-calcul. Graduate thesis, École Polytechnique, september 1999. Electronically available at `http://www.di.ens.fr/~feret/dea/dea.ps`.

[36] Jérôme Feret. Confidentiality analysis of mobile systems. In *Proc. SAS'00*, number 1824 in LNCS. Springer-Verlag, 2000.

[37] Jérôme Feret. Abstract interpretation-based static analysis of mobile ambients. In *Eighth International Static Analysis Symposium (SAS'01)*, number 2126 in LNCS. Springer-Verlag, 2001.

[38] Jérôme Feret. Occurrence counting analysis for the $\pi$-calculus. In *Proc. GETCO'00*, volume 39.2 of *ENTCS*. Elsevier Science Publishers, 2001.

[39] Jérôme Feret. Dependency analysis of mobile systems. In *Proc. ESOP'02*, number 2305 in LNCS. Springer-Verlag, 2002.

[40] Cédric Fournet. *The Join-Calculus : A Calculus for Distributed Mobile Programming*. PhD thesis, École Polytechnique, Paris, France, 1998.

[41] Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Proc. POPL'96*. ACM Press, 1996.

[42] Philippe Granger. Static analysis of linear congruence equalities among variables of a program. In *Proc. TAPSOFT'91*, LNCS. Springer-Verlag, 1991.

[43] René Rydhof Hansen, Jacob Grydholt Jensen, F. Nielson, and H. Riis Nielson. Abstract interpretation of mobile ambients. In *Proc. SAS'99*, LNCS. Springer-Verlag, 1999.

[44] Matthew Hennessy and James Riely. Resource access control in systems of mobile agents. In *Proc. HLCL'98*, volume 16.3 of *ENTCS*. Elsevier Science Publishers, 1998.

[45] Matthew Hennessy and James Riely. A typed language for distributed mobile processes. In *Proc. POPL'98*. ACM Press, 1998.

[46] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the $\pi$-calculus. In *Proc. POPL'01*. ACM, 2001.

[47] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 1976.

[48] Cosimo Laneve and Björn Victor. Solos in concert. In *Proc. ICALP'99*. Springer-Verlag, 1999.

[49] Cosimo Laneve and Björn Victor. Solos in concert. *Mathematical Structures in Computer Science*, 13(5), 2003.

[50] Francesca Levi and Sergio Maffeis. An abstract interpretation framework for analysing mobile ambients. In *Proc. SAS'01*, volume 2126 of *LNCS*. Springer-Verlag, 2001.

[51] Francesca Levi and Sergio Maffeis. On abstract interpretation of mobile ambients. *Information and Computation*, 188(2), 2004.

[52] Francesca Levi and Davide Sangiorgi. Controlling interference in ambients. In *Proc. POPL'00*. ACM Press, 2000.

[53] Laurent Mauborgne. ASTRÉE : Verification of absence of run-time error. In René Jacquart, editor, *Building the information Society (18th IFIP World Computer Congress)*, pages 384–392. The International Federation for Information Processing, Kluwer Academic Publishers, Aug 2004.

[54] Robin Milner. The polyadic $\pi$-calculus : a tutorial. In *Proceedings of the International Summer School on Logic and Algebra of Specification*. Springer-Verlag, 1991.

[55] Antoine Miné. The octagon abstract domain. In *Proc. AST'01 in WCRE'01*. IEEE Press, 2001.

[56] Antoine Miné. A few graph-based relational numerical abstract domains. In *Proc. SAS'02*, LNCS. Springer-Verlag, 2002.

[57] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proc. ESOP'04*, LNCS. Springer, 2004.

[58] Antoine Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polytechnique, 2004. to appear.

[59] Flemming Nielson, Hanne Riis Nielson, René Rydhof Hansen, and Jacob Grydholt Jensen. Validating firewalls in mobile ambients. In *Proc. CONCUR'99*, LNCS. Springer-Verlag, 1999.

[60] Flemming Nielson, Hanne Riis Nielson, and Henrik Pilegaard. Spatial analysis of bioambients. In *Proc. SAS'04*, volume 3148 of *Lecture Notes on Computer Science*. Springer Verlag, 2004.

[61] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Normalizable horn clauses, strongly recognizable relations, and spi. In *Proc. SAS'02*, volume 2477 of *Lecture Notes on Computer Science*. Springer Verlag, 2002.

[62] Flemming Nielson and Helmut Seidl. Control-flow analysis in cubic time. In *Proc. ESOP'01*, LNCS. Springer-Verlag, 2001.

[63] Hanne Riis Nielson and Flemming Nielson. Shape analysis for mobile ambients. In *Proc. POPL'00*. ACM Press, 2000.

[64] Rohit J. Parikh. On context-free languages. *Journal of the ACM*, 13, 1966.

[65] Joachim Parrow and Björn Victor. The fusion calculus : Expressiveness and symmetry in mobile processes. In *Proc. LICS'98*, 1998.

[66] Andrew T. Phillips, Nabuko Yoshida, and Susan Eisenbach. A distributed abstract machine for boxed ambient calculi. In *Proc. ESOP'04*, volume 2986 of *LNCS*. Springer-Verlag, 2004.

[67] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the $\pi$-calculus. In *Proc. SAS'01*, LNCS. Springer-Verlag, 2001.

[68] Aviv Regev, Ekaterina M. Panina, William Silverman, Luca Cardelli, and Ehud Y. Shapiro. Bioambients : An abstraction for biological compartments. *Theoretical Computer Science*, 325 :141–167, 2004.

[69] Jordi Cortadella Robert Clarisó. The octahedron abstract domain. In *Proc. SAS'04*, LNCS. Springer-Verlag, 2004.

[70] Davide Sangiorgi. From $\pi$-calculus to Higher-Order $\pi$-calculus — and back. In *Proc. TAPSOFT'93*, lncs. Springer-Verlag, 1993.

[71] Davide Sangiorgi and Andrea Valente. A distributed abstract machine for Safe Ambients. In *Proc. ICALP'01*, volume 2076 of *LNCS*. Springer-Verlag, 2001.

[72] David N. Turner. *The Polymorphic $\pi$-Calculus : Theory and Implementation*. PhD thesis, Edinburgh University, 1995.

[73] Arnaud Venet. Abstract interpretation of the $\pi$-calculus. In *Proc. LOMAPS'97*, LNCS. Springer-Verlag, 1997.

[74] Arnaud Venet. Automatic determination of communication topologies in mobile systems. In *Proc. SAS'98*, LNCS. Springer-Verlag, 1998.

[75] Jan Vitek and Giuseppe Castagna. Seal : A framework for secure mobile computations. In *Proc. IPL'99*. Springer-Verlag, 1999.

[76] Thomas Y. C. Woo and Simon S. Lam. Authentication for distributed systems. In *Internet besieged : countering cyberspace scofflaws*, pages 319–355. ACM Press/Addison-Wesley Publishing Co., 1998.