



HAL
open science

Static Analysis of an Actor-based Process Calculus by Abstract Interpretation

Pierre-Loïc Garoche

► **To cite this version:**

Pierre-Loïc Garoche. Static Analysis of an Actor-based Process Calculus by Abstract Interpretation. Software Engineering [cs.SE]. Institut National Polytechnique de Toulouse - INPT, 2008. English. NNT : 2008INPT006H . tel-00310923

HAL Id: tel-00310923

<https://theses.hal.science/tel-00310923>

Submitted on 11 Aug 2008

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.



Université
de Toulouse



année : 2008
n° d'ordre : 2620

THÈSE

soutenue en vue de l'obtention du titre de

**DOCTEUR EN INFORMATIQUE
DE L'UNIVERSITÉ DE TOULOUSE**

délivré par

L'INSTITUT NATIONAL POLYTECHNIQUE DE TOULOUSE

ÉCOLE DOCTORALE MATHÉMATIQUES, INFORMATIQUE ET
TÉLÉCOMMUNICATIONS DE TOULOUSE

Pierre-Loïc Garoche

10 juin 2008

Analyse statique d'un calcul d'acteurs par interprétation abstraite

Static Analysis of an Actor-based Process Calculus by Abstract Interpretation

Directeur de thèse : Patrick SALLÉ
Professeur, INPT

Rapporteurs : Thomas JENSEN
Directeur de recherches au CNRS, IRISA
Flemming NIELSON
Professor, Technical University of Denmark

Examineurs : Jean-Jacques LEVY
Directeur de recherches à l'INRIA
Marc PANTEL
Maître de conférence, INPT
Xavier THIRIOUX
Maître de conférence, INPT
Virginie WIELS
HDR, Ingénieur de recherches à l'ONERA

REMERCIEMENTS

Formuler ses remerciements est une étape obligée dans la rédaction d'une thèse. Je n'y couperai donc pas. Ces remerciements s'adressent à l'ensemble des personnes qui ont participé, de près ou de loin, à l'aboutissement de cette thèse. Je tiens par avance à m'excuser de mes oublis éventuels.

Mes remerciements vont tout d'abord à Flemming NIELSON et Thomas JENSEN qui ont bien voulu accepter de relire mon manuscrit. Leur point de vue éclairé ainsi que leur compétence donnent, à mes yeux, du crédit à mon travail, et je les en remercie. I want to particularly thank Flemming NIELSON and the members of his group in Copenhagen, for their great welcome during my very short stay at DTU. They showed a deep interest in my work during the not-so-funny time of writing the manuscript. It gave me a second breath and help me to continue and regain self-confidence.

Je tiens également à remercier les autres membres du jury qui ont accepté d'évaluer mon travail. Virginie WIELS a un lien spécial avec ce calcul d'acteurs que j'analyse et Jean-Jacques LEVY a été l'un des enseignants de mon master et a beaucoup contribué à la définition et l'analyse de calculs de processus. J'apprécie beaucoup qu'ils aient accepté de participer à mon jury.

Je veux adresser un remerciement particulier à Patrick COUSOT, qui m'a initié durant mon master aux arcanes de son Interprétation Abstraite et m'a passionné pour sa méthode. C'est également à lui que je dois mon arrivée à Toulouse. Pour cela, merci.

Merci à Patrick SALLÉ et Marc PANTEL qui m'ont ensuite accueilli à Toulouse dans leur équipe. Ils m'ont laissé toute liberté dans mes recherches tout en me donnant la possibilité de participer à des congrès. Merci de m'avoir également permis d'effectuer des enseignements passionnants au sein du département informatique de l'ENSEEIH. Je tiens tout particulièrement à remercier Xavier THIRIOUX pour son aide précieuse apportée lors de la définition des analyses, de leur preuve et de leur implantation.

Lors de la réalisation de cette thèse, j'ai pu profiter du cadre agréable de l'IRIT-N7. Je tiens à remercier dans le désordre, les membres de l'équipe, en particulier Philippe QUEINNEC et Gérard PADIOU, pour le stock monstreux de chocolat qu'ils ont apporté; les amis du 4ième étage; mes co-bureaux, le professeur Ibrahim LOKPO, Aurélie HURAUULT et Ahmed TOUHAMI pour les bons moments partagés; Benoît COMBEMALE et Xavier CRÉGUT, pour m'avoir initié et fait découvrir un domaine dont j'ignorais tout : l'ingénierie dirigée par les modèles, et pour les collaborations partagées; et bien sûr, le grand chef Michel et les secrétaires exceptionnelles du laboratoire, Sylvie E., Joyce, Sabyne et Sylvie A., merci pour votre soutien et votre bonne humeur.

Je ne serais pas arrivé là non plus sans un long parcours informatique. Je tiens d'abord à remercier mon père pour m'avoir initié très tôt aux joies de la programmation et m'avoir donné le goût de la compréhension des choses. Mes premiers balbutiements informatiques avec Matthieu SOZEAU, mes études à Orsay avec les excellents cours d'Alain DENISE, Nicole BIDOIT, Jean-Pierre JOUANNAUD ou de Christine PAULIN-MOHRING, le stage effectué auprès de Marie-Claude GAUDEL, ou les mémorables moments passés à BlueBird : toutes ces rencontres m'ont donné le goût de la recherche en informatique et des méthodes formelles, merci à tous.

Un immense merci à mes collègues de Cachan, Malo, Nicolas et Sylvain qui m'ont très bien accueilli à l'ENS et avec qui l'année de master a été passionnante. Hubert COMON et Catherine FORESTIER, du département informatique de l'ENS Cachan, ont été d'un grand secours, toujours efficaces et accessibles. Merci ! Cette année 2005 fut très riche en rencontres et en connaissances acquises qui seraient trop longues à énumérer. J'adresse tout de même un grand merci à Jérôme FERET pour les échanges fructueux pendant mon master et à Xavier ALLAMIGEON pour son amitié.

Je vais arrêter cette énumération qui ne saurait être exhaustive. Laissez moi seulement remercier le professeur HONIDEN du NII et Cyrille ARTHO pour leur accueil de 3 mois à Tokyo ainsi que Christian SOMMER pour les jours et les nuits de travail acharné et de discussions enflammées partagés.

Enfin comment terminer sans remercier ma famille, mes parents, mon frère et ma belle-famille qui ont su m'apporter leur soutien ou me permettre de faire des pauses et bien sûr, Pascaline qui a su toujours m'aider et m'a donné mon plus beau cadeau pendant cette période de thèse : mon fils, Samuel. Merci !

ABSTRACT

The Actor model, introduced by HEWITT and AGHA in the late 80s, describes a concurrent communicating system as a set of autonomous agents, with non uniform interfaces and communicating by the use of labeled messages. The CAP process calculus, proposed by COLAÇO, is based on this model and allows to describe non trivial realistic systems, without the need of complex encodings. CAP is a higher-order calculus: messages can carry actor behaviors. Multiple works address the analysis of CAP properties, mainly by the use of inference-based type systems using behavioral types and sub-typing.

Otherwise, more recent works, by VENET and later FERET, propose the use of abstract interpretation to analyze process calculi. These approaches allow to compute non-uniform properties. For example, they are able to differentiate recursive instances of the same thread.

This thesis is at the crossroad of these two approaches, applying abstract interpretation to the analysis of CAP. Following the framework of FERET, CAP is firstly expressed in a non standard form, easing its analysis. The set of reachable states is then over-approximated via a sound by construction representation within existing abstract domains.

New general abstract domains are then introduced in order to improve the accuracy of existing analyses or to represent local properties.

CAP specific properties such as the linearity of terms or the absence of orphan messages, are then considered in this framework. Specific abstract domains are defined and used to check these properties. The proposed framework is able to relax any existing restriction of previous analyses such as constraints on the shape of terms or limitation in the use of CAP behavior passing.

The whole analyses have been implemented in a prototype.

KEYWORDS Actors, Higher-order process calculus, Abstract Interpretation, Linearity, Orphan messages, Message counting.

RÉSUMÉ

Le modèle des Acteurs, introduit par HEWITT et AGHA à la fin des années 80, décrit un système concurrent comme un ensemble d'agents autonomes au comportement non uniforme et communiquant de façon point-à-point par l'envoi de messages étiquetés. Le calcul CAP, proposé par COLAÇO, est un calcul de processus basé sur ce modèle qui permet de décrire sans encodage complexe des systèmes réalistes non triviaux. Ce calcul permet, entre autre, la communication de comportements via les messages et est, en ce sens, un calcul d'ordre supérieur. L'analyse de propriétés sur ce calcul a déjà fait l'objet de plusieurs travaux, essentiellement par inférence de type en utilisant des types comportementaux et du sous-typage.

Par ailleurs, des travaux plus récents, effectués par VENET puis FERET, proposent une utilisation de l'interprétation abstraite pour l'analyse de calculs de processus. Ces approches permettent de calculer des propriétés non uniformes : elles permettent, par exemple, de différencier les instances récursives d'un même processus.

Cette thèse s'inscrit donc dans la suite de ces deux approches, en appliquant l'interprétation abstraite à l'analyse de CAP. Suivant le cadre proposé par FERET, CAP est, tout d'abord, exprimé dans une forme non standard facilitant les analyses. L'ensemble des configurations atteignables est ensuite sur-approximé via une représentation, correcte par construction, dans des domaines abstraits.

Des domaines abstraits généraux sont ensuite introduits afin d'améliorer les analyses existantes ou de représenter des propriétés locales à un sous-terme.

Des propriétés spécifiques à CAP, la linéarité des termes et l'absence de messages orphelins, sont alors étudiées dans ce cadre. Des domaines spécifiques sont définis et utilisés pour vérifier ces propriétés. Le cadre présenté permet de lever toutes les restrictions existantes des analyses précédentes quant à la forme des termes ou l'utilisation du passage de comportement.

L'intégralité des analyses présentées a été implantée dans un prototype.

MOTS CLÉS Acteurs, Calcul de processus d'ordre supérieur, Interprétation abstraite, Linéarité, Messages orphelins, Dénombrement de messages.

CONTENTS

I	ÉPITOMÉ	1
1	ANALYSE STATIQUE D'UN CALCUL D'ACTEURS PAR INTERPRÉTATION ABSTRAITE	3
1.1	Introduction & contexte	3
1.2	CAP : un calcul d'acteurs primitif	5
1.3	Analyse statique de CAP par interprétation abstraite	6
1.4	Sémantique non standard	6
1.5	Sémantique abstraite	7
1.6	Analyse partitionnée	10
1.7	Amélioration des analyses de dénombrement	11
1.8	La propriété de linéarité	12
1.9	Garantir l'absence de messages orphelins	13
1.10	Réalisation logicielle	13
1.11	Conclusion	14
II	MAIN CONTENT	17
2	INTRODUCTION & BACKGROUND	19
2.1	CAP: a primitive actor calculus	20
2.1.1	A bit of history	20
2.1.2	Syntax	22
2.1.3	Semantics	24
2.1.4	Examples	24
2.2	Static analysis methods	28
2.2.1	Type system-based	28
2.2.2	Model checking	30
2.2.3	Abstract interpretation	31
2.2.4	Flow logic	35
2.3	Concurrency analysis related works	35
2.3.1	Typing process calculi	35
2.3.2	Flow logic analysis of concurrency	36
2.3.3	Model checking and concurrency	36
2.3.4	Abstract interpretation-based analysis of process calculi	37
2.4	Overview of contributions	37
2.4.1	CAP non standard semantics	38
2.4.2	Abstracting collecting semantics and abstract domains	38
2.4.3	Linearity	38
2.4.4	Orphan freeness checking	39

	2.4.5	Implementation issues	39
3		STATIC ANALYSIS OF CAP BY ABSTRACT INTERPRETATION	41
	3.1	The intuition	41
	3.1.1	FERET's framework	41
	3.1.2	Instantiating the framework to model CAP semantics	43
	3.2	Instantiating the generic framework	44
	3.2.1	Generic framework semantics	44
	3.2.2	Partial interactions	44
	3.2.3	Formal rules	46
	3.2.4	Syntax extraction	47
	3.2.5	Operational semantics	50
	3.2.6	Resulting transition system	52
	3.2.7	Soundness	53
	3.3	Abstracting non standard semantics	54
	3.3.1	Abstracting collecting semantics	55
	3.3.2	Approximating control flow	56
	3.3.3	Occurrence counting	61
	3.4	Discussion	66
4		PARTITIONED ABSTRACT DOMAIN	71
	4.1	Partitioning properties by address	72
	4.1.1	Concrete address partitioning	72
	4.1.2	Abstract address partitioning	77
	4.2	Parametric Abstract Partitioning	80
	4.2.1	Intuition	80
	4.2.2	Abstract Domain	81
	4.2.3	Semantics primitives	82
	4.2.4	Operational semantics	87
	4.3	Example analysis	90
	4.4	Enhancement	92
	4.4.1	Extending primitives	92
	4.4.2	Soundness	95
	4.4.3	Application	95
	4.5	Related work and discussion	96
	4.5.1	Comparison with Feret's thread partitioning	96
	4.5.2	Summary	97
5		ENHANCING OCCURRENCE COUNTING	99
	5.1	The initial occurrence counting abstraction	99
	5.1.1	Numerical abstractions	100
	5.1.2	Example analysis	107
	5.2	Enhancing the abstraction reduction	109
	5.2.1	Motivation	109
	5.2.2	The reduction revisited	111
	5.2.3	The example reconsidered	112

5.3	Considering computed transitions	112
5.3.1	Motivation	112
5.3.2	Abstract domain	114
5.3.3	The example reconsidered	116
5.4	Reduction in the partitioned domain	117
5.4.1	Combining abstract domains	118
5.4.2	Reduction	118
5.5	Summary	119
5.5.1	Contributions	119
5.5.2	General overview	120
6	THE LINEARITY PROPERTY	123
6.1	Problematics	123
6.1.1	Definition	124
6.1.2	Examples	124
6.1.3	A first attempt of linearity checking in our framework	124
6.2	Abstracting linearity	127
6.2.1	Intuition	127
6.2.2	A first abstraction	128
6.2.3	A second abstraction	134
6.3	Example analysis	138
6.4	Related works	138
6.5	Discussion	140
7	ENSURING ORPHAN FREENESS	143
7.1	Problematics	143
7.1.1	Definitions	144
7.1.2	Examples	147
7.2	Roadmap to orphan freeness checking	149
7.2.1	Observation	149
7.2.2	Vector Addition System with States and their properties	150
7.2.3	Effective checking	151
7.3	Abstracting mailboxes and interfaces	152
7.3.1	Intuition	152
7.3.2	Abstract domain	154
7.3.3	Improvements preserving soundness	162
7.4	Ensuring orphan-freeness	164
7.4.1	On effective mailbox computation	164
7.4.2	Where over-approximations comes from	166
7.4.3	Checking orphan-freeness: under-approximating interfaces	168
7.4.4	Checking non stuck actors	168
7.5	Example analysis	169
7.5.1	Example	169
7.5.2	Resulting abstract properties	169
7.5.3	Computing mailboxes	170

7.5.4	Checking orphan-freeness	170
7.6	Related works and discussion	171
7.6.1	Analyzing CAP by type inference	171
7.6.2	Behavioral types for the π -calculus	172
7.6.3	Encoding properties into VASS-like structures	172
7.6.4	Discussion	173
8	IMPLEMENTATION ISSUES	175
8.1	PACSA: a Primitive Actor Calculus Static Analyzer	175
8.1.1	Wide use of Caml modules and functors	176
8.1.2	Implementation choices	179
8.1.3	Domains	181
8.2	Results	183
8.3	Usage	184
8.3.1	Command line	185
8.3.2	Web interface	187
9	CONCLUSION	191
9.1	Contributions	192
9.2	Future Works	194
9.2.1	Implementing mailboxes over-approximation	194
9.2.2	Introducing relational abstraction in the linearity abstract domain	195
9.2.3	Applying the proposed domains to the analysis of π -calculus	195
9.2.4	Analyzing other kinds of concurrent communicating models	195
9.2.5	Weaving abstract interpretations	196
	INDEX OF NOTATIONS	199
	BIBLIOGRAPHY	203
III	APPENDICES	213
A	PROOFS	215
A.1	Bisimulation between CAP semantics and its non standard encoding	215
A.2	Partitioned abstract domain	221
A.3	Occurrence counting with transitions	227
A.4	Linearity	229
A.5	Orphan messages	231
B	REPLICATING EXAMPLE ANALYSIS	237
B.1	Replication server example	237
B.2	Analysis	237
B.2.1	Control flow abstraction	237
B.2.2	Global occurrence counting abstraction	241

B.2.3	Linearity abstraction	242
B.2.4	Partitioned abstraction	243

LIST OF FIGURES

Figure 2.1	CAP syntax.	23
Figure 2.2	CAP transition relation.	25
Figure 2.3	CAP structural context rules.	25
Figure 2.4	CAP congruence relation.	25
Figure 2.5	A simple toy example.	26
Figure 2.6	A simple behavior passing.	27
Figure 2.7	Non trivial non linearity.	27
Figure 2.8	Linear cell.	28
Figure 2.9	Replicating server.	29
Figure 2.10	Replicated server use.	29
Figure 2.11	Framework overview.	39
Figure 2.12	Hierarchy of abstract domains.	40
Figure 3.1	Communication with a syntactically defined actor.	47
Figure 3.2	Communication with a dynamic actor.	48
Figure 3.3	Non standard operational semantics.	51
Figure 3.4	Abstract operational semantics for control flow approximation.	58
Figure 3.5	Transition computation for the abstract domain of equalities and disequalities among variables.	59
Figure 3.6	Transition computation for the abstract domain of regular approximation of shape.	62
Figure 3.7	Abstract operational semantics for occurrence counting abstraction.	65
Figure 4.1	CAP non standard partitioned semantics.	75
Figure 4.2	CAP example concrete address partitioning.	78
Figure 4.3	Partitioned abstract domain operational semantics.	89
Figure 4.4	Partitioned occurrence counting.	91
Figure 4.5	Partitioned control flow.	93
Figure 5.1	The lattice of intervals in \mathbb{N}^2 .	100
Figure 5.2	KARR's union algorithm cases.	103
Figure 5.3	Linear cell: an intermediate occurrence counting abstract element.	108
Figure 5.4	Linear cell: another intermediate occurrence counting abstract element.	110
Figure 5.5	Linear cell: an intermediate occurrence counting abstract element computing a spurious transition.	113
Figure 5.6	Abstract operational primitive for the occurrence counting domain with transitions.	117

Figure 5.7	Overview of the global occurrence counting abstraction.	121
Figure 6.1	Non linearity example trace.	125
Figure 6.2	Abstract operational semantics for first abstraction.	133
Figure 6.3	Abstract operational semantics for second abstraction.	138
Figure 6.4	Linearity: abstract properties computation.	139
Figure 7.1	Example traces with mailboxes.	148
Figure 7.2	System S_2 : an (infinite) ideal case.	150
Figure 7.3	Message sending over-approximation outside partition unit.	155
Figure 7.4	Resulting abstract mailboxes and interfaces for the linear cell-based systems.	156
Figure 7.5	Behavior passing example: resulting abstract element.	170
Figure 8.1	Overview of our analysis process.	175
Figure 8.2	Main abstract domain module type.	176
Figure 8.3	Control flow abstract domain module types: atoms and molecules.	177
Figure 8.4	Numerical abstract domain module type for the occurrence counting abstraction.	178
Figure 8.5	Generic module type for defining a calculus.	178
Figure 8.6	Hierarchy of modules and functors.	180
Figure 8.7	Command line usage.	185
Figure 8.8	Results of CAP term analysis on command line.	186
Figure 8.9	Home page of the web interface of PACSA.	187
Figure 8.10	Examples descriptions given in the web site.	188
Figure 8.11	A first computation showing a log of the analysis.	189
Figure 8.12	A second computation showing resulting abstract elements for the interfaces and mailboxes approximation.	190
Figure B.1	Interface and mailbox abstraction for the binder 1.	243
Figure B.2	Interface and mailbox abstraction for the binder 2.	244
Figure B.3	Interface and mailbox abstraction for the binder 3.	245
Figure B.4	Interface and mailbox abstraction for the binder 4.	246
Figure B.5	Interface and mailbox abstraction for the binder 13.	247
Figure B.6	Interface and mailbox abstraction for the binder 14.	248
Figure B.7	Interface and mailbox abstraction for the binder 19.	249

Part I

ÉPITOMÉ

ANALYSE STATIQUE D'UN CALCUL D'ACTEURS PAR INTERPRÉTATION ABSTRAITE

1.1 INTRODUCTION & CONTEXTE

Les systèmes informatiques sont maintenant de plus en plus présents et globaux. Ils forment de grands systèmes, exécutés sur de multiples machines en parallèle et sont conçus pour fonctionner sans limite de durée. Ils doivent fournir leur service de façon permanente. Ils sont également construits de façon incrémentale, en réutilisant et en étendant d'anciens systèmes par de nouveaux composants. Le développement de ces systèmes globaux requiert des années de développement et fait appel à de multiples programmeurs. Parmi ces grands systèmes, la suite de logiciels en ligne fournie par l'entreprise Google[®] est caractéristique : toutes les données sont stockées dans leurs serveurs distribués et répliqués, présents sur toute la planète. Le logiciel client envoie une requête pour une vue locale des données et est capable d'interagir avec le système à partir de n'importe quel périphérique connecté au réseau.

Ces systèmes que l'on peut qualifier d'*orientés services* se profilent comme la prochaine génération de logiciels, nous donnant un accès persistant à nos données. Cependant la complexité de ces systèmes est inhérente : ils sont grands, ils s'exécutent de façon concurrente, ils sont communicants et s'exécutent théoriquement pour un laps de temps infini. Un enjeu majeur actuel concerne donc la définition de techniques et de méthodes formelles permettant de vérifier ces logiciels, c'est à dire de garantir qu'ils aient un comportement cohérent et une qualité de service permanente.

Les méthodes d'analyses statiques s'intéressent au calcul de propriétés des systèmes en considérant leur description, leur code source par exemple, et en sur-approximant tous leurs comportements possibles. Elles permettent d'effectuer, de façon plus ou moins précise, des analyses qui déterminent le comportement d'un système avant son exécution. Parmi ces techniques, l'interprétation abstraite définie par Patrick COUSOT [34] est un cadre général dans lequel chaque analyse statique peut être exprimée. Cette approche a montré sa maturité et est de plus en plus utilisée dans des contextes industriels.

Concernant les systèmes concurrents et communicants, les analyses existantes sont plus simples. Elles considèrent des descriptions de très bas niveau des communications d'un système. En pratique, le non déterminisme, induit par la nature concurrente de ces systèmes, rend leurs analyses plus complexes que celles de programmes uniquement séquentiels. De la même façon, la communication et l'évolution de la topologie des échanges introduit elle aussi des difficultés

dans l'obtention de propriétés précises. Finalement, nous avons mentionné le caractère permanent de ces systèmes qui amène donc à considérer des traces infinies. Tous ces ingrédients contribuent à définir un espace d'état difficilement analysable par des techniques de vérification de modèle (model-checking) qui requièrent une représentation explicite de cet espace.

Cette thèse propose l'utilisation de l'interprétation abstraite à l'analyse d'un calcul concurrent. De telles analyses appliquées à des algèbres de processus de bas niveau, comme le π -calcul, existent, mais n'existent pas pour des calculs de plus haut niveau, permettant d'exprimer sans encodage complexe des systèmes réalistes. Dans ce document, nous considérons l'analyse d'un calcul de processus asynchrone, basé sur le modèle des Acteurs. Nous proposons des analyses à base d'interprétations abstraites afin de calculer automatiquement les propriétés des systèmes décrits dans ce calcul.

Dans ce chapitre introductif en français, nous introduisons le plan du manuscrit ainsi que les différents développements proposés. L'intégralité des analyses : définitions de domaines abstraits, résultats, théorèmes de correction, *etc.* est référencée et peut être trouvée dans la partie rédigée en anglais.

Le second chapitre, le chapitre 2, page 19 introduit l'objet de l'étude, le calcul d'acteurs CAP et présente brièvement les méthodes d'analyses statiques ainsi que leur application à l'étude de la concurrence dans les systèmes communicants.

Ensuite, le chapitre 3, page 41 présente l'utilisation du cadre proposé par FERET dans [49] pour exprimer puis analyser CAP.

Les chapitres 4, 6 et 7, respectivement pages 71, 123 et 143 présentent les différentes analyses proposées pour CAP dans un tel cadre, à savoir :

- les problématiques de dénombrement, c.-à-d. le nombre d'acteurs ou de messages dans les états accessibles du système analysé ;
- la vérification de la propriété de linéarité, qui voit chaque adresse du système comme une ressource non partageable et qui garantit qu'au plus un acteur est associé à une adresse donnée ;
- la vérification d'absence de messages orphelins. Un message orphelin est, dans CAP, un message qui est envoyé à une adresse qui ne peut et ne pourra jamais le réceptionner.

Enfin, le chapitre 8 illustre ces résultats théoriques : il présente PACSA, un analyseur statique par interprétation abstraite qui implante ces analyses. Ce chapitre présente également l'utilisation de PACSA sur un système décrit par un terme CAP.

L'ensemble des contributions de cette thèse est résumée dans les figures 2.11, page 39 et 2.12, page 40 qui présentent l'approche générale d'analyse des termes CAP et la hiérarchie de domaines abstraits utilisés ou introduits.

1.2 CAP : UN CALCUL D'ACTEURS PRIMITIF

Le calcul CAP a été introduit dans [24] afin de pouvoir définir des analyses statiques sur un calcul formel modélisant le modèle de la programmation par acteurs. Les termes de ce calcul décrivent des systèmes *asynchrones* composés d'*acteurs* et de *messages* étiquetés. Un acteur ($a \triangleright P$) peut être vu comme un lien entre une adresse et un comportement. Le comportement est la capacité à comprendre un certain nombre de messages ainsi que la réaction effectuée lors de la prise en compte de ces messages. L'opérateur ζ , inclus dans les sous-termes de comportement, permet de désigner dans la continuation l'adresse et le comportement total de l'acteur recevant le message. CAP permet donc de décrire des comportements dans les variables et d'envoyer celles-ci. C'est donc un calcul de processus de second ordre. La syntaxe et la sémantique de CAP sont décrites dans la figure 1. Un exemple illustrant le calcul est donné dans la figure 2.

$$\begin{array}{l}
C ::= 0 \mid \nu a^\alpha C \mid C \parallel C \mid a \triangleright^l P \mid a \triangleleft^l m(\tilde{P}) \\
P ::= x \quad \mid \quad [m_i^{l_i}(\tilde{y}) = \zeta(e, s)C_i^{i=1\dots n}] \\
\\
T = [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}] \left\{ \begin{array}{l} m = m_k, \\ |\tilde{y}_i| = |\tilde{x}_k|, \\ k \in [1, \dots, n] \end{array} \right. \\
\hline
a \triangleright T \parallel a \triangleleft^l m(\tilde{y}_i) \xrightarrow{(l, l_i, l_k)} C_k[e_k \leftarrow a, s_k \leftarrow T, \tilde{x}_k \leftarrow \tilde{y}_i]
\end{array}$$

Figure 1: Syntaxe et sémantique de CAP

Nous nous intéressons à la vérification par analyse statique de systèmes distribués décrits dans CAP. Les propriétés strictement liées à l'aspect concurrent et distribué de ces systèmes, que nous voulons ici vérifier, sont par exemple la propriété de linéarité ou la détection de messages orphelins. La première considère chaque adresse comme une ressource et consiste à vérifier qu'au plus un acteur est associé à une même adresse dans tous les termes atteignables à partir du terme initial. Les orphelins sont les messages qui sont envoyés à une adresse qui ne pourra pas les traiter.

Des travaux précédents [23, 30, 39] utilisaient de l'inférence de types afin de vérifier ces propriétés ; mais elles rencontraient des difficultés dans l'analyse de ces propriétés complexes sur CAP, un calcul d'ordre supérieur puisque qu'il permet d'envoyer des comportements (des fonctions) via des messages. L'objectif premier de ces travaux a donc été d'analyser ces propriétés en changeant de méthode d'analyse et en utilisant le cadre de l'interprétation abstraite.

$$\begin{aligned}
& \nu a^\alpha, b^\beta, \quad a \triangleright^1 [m^2() = \zeta(e, s)(a \triangleright^3 s), \\
& \quad \text{send}^4(x) = \zeta(e, s)(x \triangleleft^5 \text{beh}(s))] \\
& \quad \parallel b \triangleright^6 [\text{beh}^7(x) = \zeta(e, s)(e \triangleright^8 x)] \\
& \quad \parallel a \triangleleft^9 \text{send}(b) \parallel b \triangleleft^{10} m()
\end{aligned}$$

Deux acteurs et deux messages sont présents dans le système. Initialement, seul le message `send` envoyé à `a` peut être compris. La transition s'effectue. Il n'y a plus qu'un acteur sur `b` et deux messages adressés à cette adresse. Le message `m` ne peut toujours pas être compris, mais le nouveau message `beh` peut l'être. Il est alors reçu. La variable `s`, argument du message, dénote tout le comportement de l'acteur initial sur `a`. Ce comportement est utilisé pour être associé à l'adresse décrite par `e` au point de programme 7, c'est à dire `b`. L'acteur sur `b` peut maintenant comprendre en compte les messages `m` et `send`. Il peut alors recevoir le premier message `m`.

Figure 2: Exemple d'un système décrit par un terme CAP

1.3 ANALYSE STATIQUE DE CAP PAR INTERPRÉTATION ABSTRAITE

Afin de proposer un calcul commun pour définir des analyses génériques sur plusieurs calcul de processus, FERET exploite dans [49] un métalangage qui explicite l'historique des transitions conduisant à la création des valeurs et des agents. Une telle approche permet d'éviter les ambiguïtés liées à l' α -conversion dans l'analyse des termes. Le chapitre 3 décrit comment donner à CAP une telle sémantique puis explicite les analyses existantes d'un tel système.

1.4 SÉMANTIQUE NON STANDARD

Une configuration est maintenant un ensemble de processus. Les processus permettent de désigner les messages, les acteurs et enfin les comportements atomiques (chaque branche d'un ensemble de comportements associé à un acteur). Un processus est défini par un triplet (p, id, E) où p est le point de programme permettant d'associer le processus à un sous-terme du terme initial, id est le mot désignant la suite de transition ayant conduit à la création du processus. Enfin E , l'environnement, est une fonction partielle des variables du terme vers leur valeur. La valeur associée à une variable est un couple (α, id_α) où α est le point de programme de l'opérateur de restriction (ν ou ζ) ayant défini le nom et id_α le mot qui représente la série de transitions ayant conduit à la création de cette valeur.

Nous décrivons maintenant les étapes nécessaires au calcul d'une transition. Il faut tout d'abord choisir un *ensemble de processus* participant à la transition dans l'ensemble des processus qui constituent la configuration courante.

Il faut ensuite vérifier des *conditions de synchronisation*. En particulier, nous vérifions que le processus représentant le message est bien envoyé à l'adresse associée au processus qui dénote l'acteur. La vérification de ces conditions dépend du domaine utilisé pour représenter le flot de contrôle.

Si le n-uplet de processus sélectionné satisfait aux conditions, la transition peut s'effectuer, il faut *retirer l'ensemble des processus qui a interagi* et calculer les *nouveaux processus à insérer* dans la configuration. Pour cela, la continuation associée au comportement qui a été activé par le message est mise à jour à la fois par les valeurs contenues dans le message mais aussi par l'opérateur ζ , c'est à dire, par les valeurs décrivant l'adresse de l'acteur inter-agissant et son comportement.

Afin d'exprimer CAP dans le cadre générique proposé par FERET [49], nous avons défini un certain nombre de primitives :

- un ensemble d'interactions partielles décrivant les différentes sortes de processus, les messages, les acteurs et les comportements ;
- une fonction d'extraction β construit la configuration non standard associée à un terme CAP et est utilisée notamment lors du calcul de continuation pour déterminer les processus nouveaux à insérer dans le système ;
- enfin deux *règles formelles* ou règles de réduction permettant de modéliser dans la sémantique non standard les transitions. La première relation de transition permet de représenter une transition faisant intervenir un acteur défini syntaxiquement ($a \triangleright [..]$) tandis que la seconde nécessite trois processus et représente un acteur dont le comportement est défini par une variable ($a \triangleright b$).

La sémantique non standard est décrite dans la figure 3.3, page 51.

Théorème 1.1 (correspondance) *La sémantique standard et non standard de CAP sont en bisimulation forte.*

Ce résultat permet de garantir un correspondance, une congruence structurale entre les deux modèles : un terme CAP d'une part et son encodage dans la forme non standard d'autre part.

La preuve est présentée dans l'annexe A.1, page 215.

1.5 SÉMANTIQUE ABSTRAITE

L'expression d'un terme CAP dans cette syntaxe et cette sémantique non standard a permis d'explicitier certaines ambiguïtés dues, entre autre, à la règle d' α -conversion de la relation de congruence définissant la sémantique du langage. Le même chapitre 3 introduit comment sur-approximer la sémantique collectrice des termes exprimés dans cette sémantique non standard. Nous utilisons

pour cela le cadre de l'interprétation abstraite, afin de définir une approximation correcte et décidable, c'est à dire ici, calculable en un temps fini, de la sémantique collectrice du terme analysé.

Interprétation abstraite

L'interprétation abstraite [34, 35] est une théorie de l'approximation discrète de sémantique. La description des propriétés d'une sémantique, dans un treillis complet munis d'opérateurs monotones, permet de définir une correspondance de Galois (c'est à dire, un couple de morphisme (α, γ)) entre deux descriptions de sémantiques. La première étant la sémantique concrète (C, \sqsubseteq) , que nous voulons analyser, et la seconde une sémantique abstraite $(C^\#, \sqsubseteq)$, dans laquelle nous observons les propriétés. La correspondance de Galois (α, γ) , exhibée entre ces deux sémantiques, permet de garantir la correction de l'abstraction : c'est une sur-approximation de la sémantique concrète. Ainsi, $\forall x \in C, \forall y \in C^\#, \alpha(x) \sqsubseteq y \implies x \sqsubseteq \gamma(y)$. L'utilisation d'opérateurs monotones sur ces treillis permet de conserver cette propriété. Enfin, les points fixes pour ces opérateurs monotones existent (TARSKI) et peuvent être obtenus de façon constructive en itérant à partir d'un élément du treillis (KLEENE, COUSOT). Les propriétés étant approchées, représentées dans des domaines abstraits, nous ne pouvons répondre de façon automatique à toutes les questions, mais si la propriété étudiée peut être observée dans l'abstrait, elle est valide dans le concret. Le cadre de l'interprétation abstraite permet donc de sur-approximer de façon correcte, mais généralement pas complète, les propriétés d'une sémantique.

Nous voulons ici vérifier des propriétés sur l'ensemble des termes atteignables, c'est à dire constructibles à partir du terme initial. C'est la sémantique collectrice du terme. Elle peut être définie plus formellement comme le plus petit point fixe du morphisme complet pour l'union \mathbb{F} :

$$\mathbb{F}(X) = \left\{ \begin{array}{l} (\{\epsilon\} \times \mathcal{C}_0) \cup \\ \left\{ (u, \lambda, C') \mid \exists C \in \mathcal{S}, (u, C) \in X \text{ et } C \xrightarrow{\lambda} C' \right\} \end{array} \right\}$$

où \mathcal{C}_0 désigne la configuration initiale et \mathcal{S} l'ensemble des configurations.

Nous voulons vérifier des propriétés sur l'ensemble des ces configurations atteignables. Mais cet ensemble n'est, en général, pas calculable. Nous devons donc recourir à des méthodes permettant de vérifier ces propriétés, tout en assurant d'obtenir une réponse au bout d'un temps fini raisonnable.

L'interprétation abstraite de systèmes mobiles, définie par FERET [49], consiste à définir une sémantique opérationnelle abstraite des termes. L'isomorphisme $\mathbb{F}^\#$ sur $C^\#$ permettant d'approximer \mathbb{F} est défini par :

$$\mathbb{F}^\#(c^\#) = \bigsqcup^\# \left(\begin{array}{l} \{C_0^\#; c^\#\} \sqcup \\ \{c'^\# \mid \exists \lambda \in \Sigma, c^\# \rightsquigarrow^\lambda c'^\#\} \end{array} \right)$$

où $C_0^\#$ est l'abstraction de l'état initial et \rightsquigarrow représente la fonction de transition abstraite. Nous calculons donc l'ensemble des propriétés représentées dans le domaine abstrait $C^\#$ qui sont vraies dans un sur-ensemble de toutes les configurations atteignables. La précision de ce domaine abstrait permet donc de calculer à la fois un sur-ensemble le plus précis possible (le plus petit) et en même temps les propriétés d'intérêt.

La section suivante adresse ce problème et décrit quelle sémantique abstraite donner aux termes de la sémantique non standard et comment y représenter les propriétés qui nous intéressent.

Domaines abstraits

La sémantique opérationnelle non standard permet de guider les transitions et donc de décrire l'évolution du terme. Nous voulons ici représenter dans les domaines abstraits, à la fois l'information utile aux calculs des transitions ainsi que l'information nécessaire à la vérification des propriétés qui nous intéressent.

Un élément abstrait, représentant un ensemble de configurations concrètes, est donc constitué de deux éléments, formant un couple (CF, P) où CF représente une approximation des valeurs des variables permettant d'effectuer les transitions, tandis que P représente un ensemble de propriétés réalisées par l'ensemble des configurations que cet élément abstrait représente.

La précision de ces deux éléments, ou plutôt des domaines auxquels ils appartiennent, ainsi que des sémantiques associées, permettent de définir d'une part, le flot de contrôle, c'est à dire l'ensemble des transitions effectuées, et d'autre part la ou les propriétés à vérifier. L'utilisation de domaines simples pour représenter le flot de contrôle permet de calculer plus rapidement le point fixe de l'analyse, mais l'élément abstrait obtenu va potentiellement représenter plus de configurations concrètes que possible ; certaines de ces fausses configurations concrètes représentées ne vont pas satisfaire la propriété étudiée et donc empêcher de la vérifier dans l'abstrait.

De la même façon pour l'élément abstrait qui représente les propriétés, un domaine simple va permettre de converger rapidement, mais ne permettra peut-être pas d'observer une propriété plus complexe.

ABSTRACTION DU FLOT DE CONTRÔLE Le flot de contrôle permet de calculer les transitions dans l'abstrait. Nous utilisons ici un domaine abstrait paramétrique dont la sémantique est très proche de la sémantique non standard. Lors du calcul d'une transition dans l'abstrait, nous vérifions d'abord que les conditions de synchronisation, telles qu'elles sont représentables dans le domaine abstrait peuvent être satisfaites. Ensuite, le passage de valeur est calculé afin de créer les nouveaux processus. Nous déterminons alors l'union de l'élément abstrait obtenu avec l'élément initial afin d'obtenir les propriétés, représentées

dans le domaine, qui sont réalisées à la fois avant et après la transition. La suppression des processus inter-agissants n'est pas représentée ici puisqu'elle ne modifie pas la valeur des marqueurs des processus ni leurs variables. Une représentation plus formelle de cette sémantique est définie dans la figure 3.4.

Le domaine utilisé pour représenter le flot de contrôle peut être ensuite choisi afin d'avoir une analyse plus ou moins précise. Un domaine très simple permet, par exemple, d'avoir une sur-approximation similaire à celle que nous obtenions en utilisant des techniques de typage : il s'agit d'abstraire complètement le marqueur des processus et des variables. Une configuration est alors un n -uplet d'environnements indicés par les points de programme du terme. Chaque environnement associe à chaque variable l'ensemble des restrictions qui ont créé la valeur. Ainsi, nous vérifions, comme dans nos systèmes de type, que le message envoyé au nom a est bien reçu par l'acteur associé au même nom a , sans tenir compte des instances récursives du même nom.

Par contre, le domaine utilisé peut être plus complexe, par exemple le produit réduit de plusieurs autres domaines [53]. Il permet alors d'avoir une analyse précise et contenant des informations relationnelles entre les valeurs des variables. Par exemple, nous pouvons obtenir que le marqueur associé à une certaine variable est le même que celui associé à une autre, lorsqu'elle partage la même valeur. Plus le domaine sera précis, plus l'abstraction sera fine et la sur-approximation petite. Il y aura donc moins de fausses configurations concrètes représentées par l'élément abstrait.

ABSTRACTION DES PROPRIÉTÉS La partie de l'élément abstrait associée aux propriétés à analyser doit être munie d'une sémantique permettant de représenter dans le domaine l'évolution de cette propriété lors de l'exécution d'une transition dans le concret. L'idée générale consiste à dire que cet élément représente les propriétés réalisées par un ensemble de configurations concrètes. Le calcul, dans l'abstrait, d'une transition, doit refléter l'évolution de la propriété par la transition. Ensuite, afin de conserver les propriétés de monotonie sur les primitives du domaine abstrait, on calcule l'union des propriétés avant et après la transition, c'est à dire l'ensemble des propriétés qui sont vraies dans les deux cas.

Les chapitres suivants décrivent la définition de domaines abstraits spécifiques pour analyser différentes propriétés.

1.6 ANALYSE PARTITIONNÉE

Un premier besoin dans l'analyse de propriétés sur les systèmes décrits dans le calcul CAP est la possibilité de représenter ces propriétés par adresse. En effet, la notion d'adresse est centrale dans CAP puisque les adresses sont le lieu de rencontre entre acteurs et messages.

Les domaines abstraits existants considèrent uniquement une abstraction globale des états atteignables mais ne permettent pas de représenter les invariants, les propriétés locales de ces états.

Le chapitre 4 introduit une telle analyse en proposant un domaine dit partitionné. Ce domaine, lui même paramétré par un domaine sous-jacent permet d'exploiter ce sous domaine en représentant ses propriétés par adresses.

Un première abstraction exacte, présentée dans la section 4.1.1, permet de partitionner les termes CAP sous la forme non standard en représentant les processus, messages et acteurs associés à chaque adresse, les processus associés aux branches de comportements étant accumulés dans une partition séparée.

Ensuite, une approximation, en utilisant le domaine abstrait sous-jacent, permet d'une part, de représenter l'ensemble des processus associés à une adresse dans un élément de ce domaine, et d'autre part, de représenter dans le même élément abstrait les propriétés associées aux instances récursives d'une même adresse. En d'autres termes, si deux adresses (α, id_1) et (α, id_2) sont associées à des processus, l'élément abstrait associé au lieu α représente les propriétés vérifiées pour les deux adresses.

Le domaine abstrait de partitionnement proposé est générique et paramétré par le domaine abstrait sous-jacent et également par un domaine abstrait de flot de contrôle. Ce domaine de flot de contrôle est utilisé pour sur-approximer de façon correcte l'ensemble des partitions qui peuvent être associées à un processus.

1.7 AMÉLIORATION DES ANALYSES DE DÉNOMBREMENT

Le chapitre suivant, le chapitre 5, introduit l'analyse de dénombrement telle que définie par FERET dans [49] et propose plusieurs améliorations. Ces extensions, qui sont génériques et ne dépendent pas du cas d'analyse de CAP en particulier, permettent d'une part, d'obtenir dans l'élément abstrait résultat des informations plus précises en terme de dénombrement, mais d'autre part, améliorent également la qualité du résultat pour les autres domaines. En effet, l'analyse de dénombrement comptabilise les processus présents et peut interdire le calcul d'une transition si les processus inter-agissants ne peuvent pas être présents en même temps dans la même configuration.

L'analyse initiale proposée par FERET repose sur deux domaines abstraits numériques : un domaine non relationnel, celui des intervalles sur \mathbb{N} et un domaine relationnel, les égalités affines de KARR. Ces deux domaines forment un produit cartésien sur lequel est construite une réduction permettant d'exploiter les informations relationnelles afin de raffiner les intervalles associés aux processus. Un tel domaine permet par exemple d'observer des contraintes d'exclusion mutuelle ou de garantir que le système est borné.

La première proposition consiste à définir un étape supplémentaire dans l'algorithme de réduction entre ces deux domaines numériques. En effet, la réduction proposée n'est pas exacte, mais a un coût algorithmique cubique. En parti-

culier, toutes les informations du domaine de KARR ne sont pas utilisables dans la réduction. L'étape ajoutée, tout en restant dans la même classe de complexité, exploite plus de contraintes et permet donc d'obtenir un élément abstrait plus précis.

Une seconde proposition, tout aussi générique, permet de prendre en compte les transitions passées nécessaires à la création de processus. Ainsi dans le terme CAP suivant, l'acteur au point de programme 1 ne peut pas recevoir le message au point de programme 4.

$$a \triangleright^1 [m^2() = \zeta(e, s)(e \triangleright^3 s \parallel e \triangleleft^4 m())] \parallel a \triangleleft^5 m()$$

Ce message est produit par la consommation de l'acteur sur 1. L'analyse de flot de contrôle détecte que la transition est calculable : les adresses sont bien compatibles ainsi que les étiquettes de messages, tandis que l'analyse de dénombrement vérifie que l'acteur et le message ont bien été déjà produits. Cette seconde proposition permet d'interdire de telles transitions entre un message et un acteur qui est son ancêtre.

Finalement, la dernière proposition d'amélioration du dénombrement exploite l'analyse partitionnée sur le dénombrement et introduit une réduction entre les informations de dénombrement partitionnées et une information plus globale.

Ces différentes propositions ont été indispensable dans l'analyse des termes CAP menée dans le cadre de ce travail. Ces deux domaines de flot de contrôle et de dénombrement, bien que n'adressant pas directement les propriétés de haut niveau qui nous intéressent pour CAP, sont essentiels dans le calcul de l'approximation de la sémantique collectrice. Un dénombrement plus précis permet donc d'interdire certaines transitions et conduit, pour lui et pour les autres domaines, à des informations plus pertinentes dans l'élément abstrait résultat de l'analyse.

1.8 LA PROPRIÉTÉ DE LINÉARITÉ

La propriété de linéarité exprime le fait que dans chaque configuration, chaque adresse est associée à, au plus, un acteur. Afin de vérifier cette propriété, nous avons défini un domaine abstrait s'inspirant d'une de nos analyses par typage afin de représenter, de façon opérationnelle, le calcul de la propriété. Les détails sont présentés dans le chapitre 6.

Le domaine proposé repose sur l'association d'un mode d'utilisation à chaque variable qui sur-approxime le nombre d'utilisations de cette valeur pour installer un acteur. Une adresse utilisée strictement plus d'une fois sera associée à la valeur \top du treillis du domaine abstrait et ne permettra pas de vérifier la propriété de linéarité. Par contre, un terme dont l'élément abstrait, obtenu par l'analyse, est tel que chaque adresse et chaque variable sont associées à un mode décrivant zéro ou une utilisation, est linéaire : la sur-approximation de toutes les configurations atteignables du système satisfait la propriété de linéarité.

La sémantique de ce domaine abstrait, représentant dans l'abstrait une transition, c.-à-d. la réception d'un message par un acteur, est constitué du calcul de deux flots. Le premier flot représente le passage naturel de valeurs : la prise en compte des paramètres du message ainsi que le lancement des nouveaux processus. Le second, qui est au cœur de la vérification de la propriété de linéarité, propage en arrière les informations d'utilisation des valeurs, les adresses, aux processus inter-agissants.

1.9 GARANTIR L'ABSENCE DE MESSAGES ORPHELINS

Une seconde propriété, spécifique aux systèmes asynchrones et plus particulièrement à CAP, avec sa notion d'interfaces non uniformes, c.-à-d. pouvant évoluer avec les transitions calculées, est la détection, ou la preuve d'absence, de messages orphelins. Un message orphelin est ici un message qui est envoyé à une adresse qui ne peut ni ne pourra le prendre en compte.

Garantir l'absence de tels messages revient à considérer l'ensemble des chemins maximaux (des traces finies et infinies) du système et à vérifier que pour chacun de ces chemins, chaque message peut être consommé.

La vérification de cette propriété est effectuée en deux phases. La première consiste à sur-approximer l'ensemble de ces chemins maximaux ainsi que les messages qui peuvent être accessibles aux différents nœuds de ces chemins. Cette phase est réalisée via un domaine abstrait spécifique, présenté dans la section 7.3, qui représente, sous la forme d'un multigraphe orienté et étiqueté, la causalité, c.-à-d. les séquences d'interfaces, ainsi que les productions et consommations de messages. Ce domaine est utilisé sous l'analyse partitionnée présentée dans le chapitre 4. Nous obtenons donc, pour chaque lieu d'adresse (chaque opérateur ν), un tel graphe.

La seconde phase considère successivement chaque partition et vérifie que chaque message potentiellement présent dans une configuration peut bien être consommé dans tous les chemins maximaux à partir de cette configuration.

1.10 RÉALISATION LOGICIELLE

L'ensemble des travaux présenté à été implanté dans un interprète abstrait PACSA. Cet analyseur est codé en OCAML et utilise massivement son architecture de modules et de foncteurs. Le chapitre 8 présente l'outil ainsi que son architecture modulaire.

Le schéma général de l'analyse, décrit dans la figure 8.1 est le suivant. Le terme CAP est parsé et exprimé sous sa forme non-standard. Ensuite, le plus petit point fixe de sa sémantique collectrice est approximé par le plus petit point fixe de la sémantique collectrice abstraite, définie par la combinaison de domaines abstraits.

Dans le cadre de l'analyse des messages orphelins, une étape, supplémentaire et non implantée à ce jour, considère la représentation effective des boîtes aux lettres ainsi que la vérification de la consommation des messages en exploitant l'élément abstrait calculé.

Le chapitre 8 présente également l'utilisation de l'outil sur un exemple ainsi que son manuel d'utilisation et des captures d'écrans.

1.11 CONCLUSION

Dans cette thèse, nous avons proposé un cadre d'analyse d'un calcul d'acteur, CAP, basé sur l'interprétation abstraite.

CAP a été introduit en 1996 dans [24]. Dans cet article de premières analyses ont été définies par inférences de types. Les travaux suivants, [23, 30, 39], s'intéressent à des propriétés de plus haut niveau sur CAP comme la linéarité ou la vérification d'absence des messages orphelins. Ces travaux reposent sur une inférence de types à la $HM(X)$. Cependant, ils rencontrent plusieurs limites inhérentes à l'approche par typage : la difficulté pour traiter l'ordre supérieur, c.-à-d. le passage de comportement dans le cadre de CAP, ainsi que le dénombrement des processus ou des utilisations des adresses. Enfin, d'un point de vue plus méthodologique, l'analyse d'une nouvelle propriété requiert de reprouver la correction depuis le début.

Une autre approche de l'analyse de systèmes concurrents et communicants a été introduite par VENET [107] puis généralisée par FERET [49]. Ces techniques utilisent l'interprétation abstraite pour sur-approximer de façon correcte et dans un cadre modulaire l'ensemble des états atteignables d'un système, sa sémantique collectrice.

Les travaux présentés dans cette thèse permettent d'exploiter la seconde approche sur le calcul CAP en résolvant toutes les difficultés méthodologiques ou de précisions rencontrées par l'inférence de types. Les contributions sont les suivantes :

- Nous exprimons CAP dans une forme non standard facilitant les analyses. Nous prouvons la correction de l'encodage en exhibant une bisimulation forte entre la sémantique standard de CAP et son encodage non standard.
- Nous adaptons des domaines existants et en introduisons de nouveaux pour représenter et analyser des propriétés spécifiques à CAP. En particulier, nous améliorons l'analyse de dénombrement et proposons une analyse partitionnée pour représenter des propriétés locales, par adresse.
- Nous introduisons un domaine abstrait spécifique à la propriété de linéarité. Cette propriété exprime qu'aucune adresse n'est utilisée dans le même terme pour lier un acteur. Le domaine proposé repose sur un calcul de flot arrière pour propager l'utilisation des valeurs aux lieux d'adresse. Ce domaine se focalise sur l'utilisation des valeurs d'adresse et exploite les propriétés des autres domaines abstraits pour sur-approximer les transitions

calculables. Ainsi, il ne fait aucune hypothèse sur la forme des termes analysés et permet de traiter le passage de comportement de CAP.

- Enfin nous proposons une analyse permettant de garantir l’absence de messages orphelins. Dans ce contexte, les orphelins sont des messages qui sont envoyés à une adresse qui ne pourra jamais les prendre en compte. Les analyses précédentes pâtissaient d’un calcul de flot de contrôle imprécis et ne permettaient pas de prendre en compte le passage de comportement. Le proposition d’analyse adresse le problème sans contrainte sur la forme des termes. Le processus de vérification de la propriété est effectué en deux phases.

La première exploite le cadre de l’interprétation abstraite, défini au dessus, pour associer à chaque adresse un *Vector Addition System with States* (VASS). Ce graphe représente la causalité entre les interfaces successives associés aux acteurs de cette adresse ainsi que l’évolution des messages accessibles à chaque instant. La propriété d’absence d’orphelins est alors exprimée comme un problème de couverture sur les VASS calculés.

La seconde phase utilise l’élément calculé pour vérifier effectivement que chaque message produit peut bien être consommé dans tous les futurs possibles.

L’ensemble des ces contributions a été implanté dans l’outil PACSA présenté au chapitre 8.

Les travaux présentés sont une nouvelle contribution au développement pratique de l’interprétation abstraite sur les calculs de processus. L’auteur soutient que cette approche à l’analyse statique des systèmes concurrents et communicants est prometteuse et présente de nombreux avantages sur les autres méthodes d’analyse. En effet, du point de vue de la méthodologie, la correction par construction des domaines abstraits, ainsi que les outils de l’interprétation abstraite pour combiner les domaines permettent de définir des analyses de façon modulaire et avec précision. Du point de vue des résultats des analyses, le cadre de l’interprétation abstraite permet, en utilisant les domaines abstraits adéquats, comme les domaines relationnels, de paramétrer le coût et la précision, en particulier concernant les analyses de flot de contrôle.

Part II

MAIN CONTENT

Nowadays computer systems are more than ever omnipresent and global. They are huge systems that are executed on multiple computers and do not run for a finite duration. We expect them to provide their service permanently. Some of their parts are old ones and some components are new ones, plugged in as extensions. Global systems development involves multiple developers in different companies and runs for years. One example of these systems is provided by the Google[®] company: all the data is stored in their distributed and replicated thousand of servers all over the world. The user client software requests a local view of the data and is able to send queries from any device connected to the global network.

These service-oriented systems promise to be the next generation of software providing a persistent access to our data. However these systems are inherently complex: they are big, concurrent, communicating and theoretically run forever. So an actual concern is to define formal methods and techniques to guarantee that they live up to the expectation of their designers.

Static analysis techniques consider system descriptions, *i.e.* their source code for example, and over-approximate all the possible behaviors. They allow to perform more or less precise and efficient analyses that predict before the execution how they will behave when executed. Among these techniques, the abstract interpretation methodology defined by Patrick COUSOT in [34] is a general framework in which every static analysis could be expressed. This technique has proved to be mature and is more and more used in the industry, proving the absence of run-time error in critical embedded software such as the 300.000 lines of code of the Airbus A380 flight control system.

Considering concurrent communicating systems, existing analyses are simpler. They consider coarse and often low level descriptions of the concurrent communications in a system. They can be compared to the early typing systems of programming languages. Actually, concurrent systems are inherently more complex than sequential ones. Non determinism plays a major role and produces a large set of possible program traces. Furthermore, as mentioned above, this kind of systems usually run forever. Building the whole state space of these systems as required by model checking techniques is hardly feasible.

This thesis targets to apply abstract interpretation-based analyses to a concurrent calculus. Analyses on low level process algebra like the π -calculus already exist. But the author advocates that there is a need for analyses devoted to more high level descriptions of concurrent systems, such as the process calculus con-

sidered here. Analyses for low level languages are interesting as they address the essential problematics of the concurrent paradigm.

In this thesis we consider an asynchronous process calculus based on the actor model. This calculus eases the definition of realistic systems without a complex encoding. We then propose static analyses based on the abstract interpretation framework to automatically compute properties of the system described in the calculus.

This preliminary chapter is organized as follows. The first section, Section 2.1, introduces the object of the current study: the CAP process calculus. It is an asynchronous calculus with non uniform interfaces. This first section presents the calculus and illustrates its use through several examples. These examples are then be referenced all along this manuscript.

The following section, Section 2.2, presents an overview of the different approaches to the static analysis of systems. It outlines their pros and cons but also gives an insight of their specific requirements or properties.

The Section 2.3 considers related works; mainly applications of static analysis techniques to verification of the concurrent calculi properties.

Finally the last section, Section 2.4, outlines our contributions.

2.1 CAP: A PRIMITIVE ACTOR CALCULUS

This section introduces CAP, an actor-based process algebra. It first motivates its creation, giving an insight to the origin of concurrent process calculi. Then CAP is formally described. We first define its syntax in a π -calculus-like way, then we state its semantics introducing its transition rule. Finally we give some examples to illustrate its use.

2.1.1 *A bit of history*

Among the first models proposed to describe concurrent systems, the Actor model was introduced by HEWITT in 1973 in [60]. Then, in the 80s, process calculi, also called process algebra, were introduced like CCS [82] by MILNER and CSP [62] by HOARE. These calculi describe algebras of terms where synchronous communications are expressed as calculus computations, rewriting matching sub-terms. In the 80s, AGHA further developed the actor model in [3, 4]. The 90s have seen a lot of developments for process algebras. MILNER proposed the π -calculus which is now the principal reference for process calculi. In parallel, multiple calculi were defined, each with its own specificity: asynchronicity, specific primitives to model cryptographic protocols, or mobility to allow modeling of distributed systems, etc.

Among the calculi proposed, we focus here on CAP [24], a process calculi which is based on the actor model.

This model of actors sees a system as a set of autonomous entities cooperating over a network. These are called *actors*. They contain data and programs and communicate through an asynchronous point-to-point protocol. Each actor is associated to a queue, often called mailbox, in which asynchronous communication, *i.e.* messages, are accumulated until their consumption by their target actor. An actor is fully defined by two elements: on the one hand its *address* and on the other hand its *behavior*. This behavior specifies which message can be handled and how it is done so. When an actor handles a message it processes the following actions:

- sending a finite number of messages to actors bound on addresses it knows;
- creating a finite number of new fresh addresses and a finite number of actors;
- changing its own behavior.

Therefore, during a system evolution, an actor can handle more or less messages depending on its associated behaviors. The sequence of behaviors depends on the ordering in which it takes messages in its mailbox, no hypothesis is made on such ordering. In this model, the communication media is supposed to be safe, loss-free. We impose a fairness assumption that can be stated as follows: “a message that can be infinitely often handled, will be handled”.

In such a model, the change of behavior may update the capability of an actor, widening the set of understandable messages, but it can also completely change its behavior. In order to illustrate this mechanism, AGHA uses in [2] the example of an actor modeling a bank account that becomes a pizza delivery service after the receiving of a message. It will then not be able anymore to answer any bank-related query.

This model is also known under the name of *concurrent objects with non-uniform interfaces*. It gives a natural framework to describe and implement concurrent and distributed systems.

The calculus studied here, CAP, was introduced by COLAÇO in [24]. It is a process calculus, in the sense of the π -calculus, that formalizes the ideas of the actor model, as a process algebra. As a more formal description, it is more precisely defined than its original model by AGHA. In particular, its syntax is more permissive as it allows to build ill-formed terms or to send behaviors within messages. The objective of the analyses is then to ensure the good behavior of such systems.

Concerning asynchronous process calculi, similar mechanisms to describe message queuing are also present in the literature, for example in the HACL concurrent object language of KOBAYASHI and YONEZAWA [75], in the blue calculus proposed by BOUDOL [19] or in the Join calculus of FOURNET *et al.* [52]. We can also mention the predecessor of CAP, the Plasma-II language as introduced by SALLÉ [79].

2.1.2 Syntax

CAP is a formal process calculus. It is therefore defined as a process algebra that describes concurrent systems based on the notions presented above. It is a calculus in the sense that terms of this algebra can be reduced, similarly to usual mathematical terms. In this concurrency context, process calculi reduction denotes communications. A reduction computation produces a new term which can again be reduced. Process calculi are inherently non deterministic as they model concurrent systems in which multiple agents, actors and messages in CAP case, can communicate in parallel.

In CAP, processes are called *actors*. They communicate together using labeled *messages*. An actor that sends a message does not wait for its receiving; it can continue its execution independently of the future of the sent message. The receiver is waiting for messages and evolves once it receives something. CAP is thus an asynchronous calculus. It is opposed to the synchronous approach of concurrency where agent communication is defined as a *rendez-vous* mechanism.

As an actor-based calculus, the processes and their reductions are specified using the notion of *behavior*. An actor in a given state is associated to a given behavior, also called interface. This behavior defines the finite set of message labels that can be handled by the actor in its current state. It also describes how the reduction is computed when receiving such messages. We call *behavior branch* the specification in a behavior of one particular message label and the associated reduction. Behaviors are then sets of behavior branches.

The actor model introduced above imposes that actors specify their next behavior when communicating. This mechanism may associate successively different behaviors to the same address.

The usual replication operator of process algebra is here replaced by the ζ operator in each behavior branch. This $\zeta(e, s)$ allows to bind variables e (ego) and s (self), respectively, the address and the behavior of the interacting actor. These variables can then be used in the associated continuation C of the behavior branch used (e.g. $m(\tilde{x}) = \zeta(e, s)C$). In that sense, the only available replication is a guarded one that necessitates a transition to be enacted. It can be encoded using behavior branches of the form

$$m(\tilde{x}) = \zeta(e, s)(e \triangleright s \parallel \dots).$$

Let \mathcal{N} be a finite set of actor names, *i.e.* addresses. Let \mathcal{V} be a finite set of variables and \mathcal{M}_l be a finite set of message labels.

In order to easily reason about CAP terms, we automatically annotate them with program points. Program points are associated to sub-terms, describing name binders (ν^l), actor definitions (\triangleright^l), messages (\triangleleft^l) and behavior branches ($m^l(\tilde{y} = \dots)$).

Let \mathcal{L}_p be the finite set of program point labels and \mathcal{L}_v be the finite set of binder program point labels. Later, in order to ease describing terms and their possible interactions, we denote by $\mathcal{L}_a, \mathcal{L}_m$ and \mathcal{L}_b the program point labels of actor sub-terms, message sub-terms and behavior branches sub-terms, respectively. Similarly we denote by \mathcal{L} the set of program point labels: $\mathcal{L} = \mathcal{L}_p \cup \mathcal{L}_v$.

In the following, we denote by \tilde{x} the vector (x_1, \dots, x_n) . The ν operators bind new addresses in the configuration. Similarly the ζ operator and the message variables in each behavior branches act as name binders for the associated behavior description. For example, in the configuration $(\nu a^\alpha)C$ and in the behavior $[m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1\dots n}]$, occurrences of a in C , variables of \tilde{x}_i in $\zeta(e_i, s_i)C_i$ and e_i and s_i in C_i are bound. α and l_i are automatically associated to sub-terms and allow to describe binders and interacting sub-terms, respectively.

Let $\mathcal{FN}(C)$ be the set of free names in the configuration C . In this document we consider only closed terms C : $\mathcal{FN}(C) = \emptyset$

The syntax of CAP terms is defined in Figure 2.1.

Figure 2.1 CAP syntax.

a	$\in \mathcal{N}$	m, m_i	$\in \mathcal{M}_l$
e_i, s_i, x, e, s	$\in \mathcal{V}$	l, l_i	$\in \mathcal{L}_p$
\tilde{x}_i	$\in \mathcal{V}^*$	α, α_i	$\in \mathcal{L}_v$
$C ::= \emptyset$			
$\nu a^\alpha C$			
$C \parallel C$			
$a \triangleright^l B$			
$a \triangleleft^l m(\tilde{P})$			
$e \triangleright^l B$			
$e \triangleleft^l m(\tilde{P})$			
$B ::= s$			
$[m_i^{l_i}(\tilde{x}) = \zeta(e_i, s_i)C_i^{i=1\dots n}]$			
$P ::= a$			
e			
B			

2.1.3 Semantics

The operational semantics of CAP which defines term evolutions is defined “à la Milner”, as usually in process calculi:

- by a transition relation, given in Figure 2.2, that describes a single actor receiving a message,
- some context rules, given in Figure 2.3, that allow a reduction to occur in a sub-part of a term, leaving the non communicating part of the term untouched,
- a congruence relation, allowing to rearrange a term to match the pattern described by the transition relation. It is given in Figure 2.4.

The transition relation expresses an actor receiving a message. It specifies the behavior change of the actor as well as some potential messages sendings or actors launching. In the following, we label such transitions by triples denoting the matching part of the term. This label is not used in the transition computation but allows us to easily describe the transition. A transition label describes the actor receiving the message, its behavior branch used and the message received.

The transition can occur iff both the actor and the message are associated to the same address, described here by the formal parameter a . The message label must be part of the finite set of message labels handleable by the actor. Lastly, the number of message arguments must match the number of formal parameters associated to the behavior branch.

The reduced term is obtained by removing both the matched actor and message from the considered term and by introducing new agents (actors and messages) described in the matched behavior branch continuation sub-term (C_k). The formal parameters used in these agents that were not previously defined, *i.e.* the variable bound by the ζ operator and the parameters of the message, are defined by value passing.

In order to ease the readability of the transition rule of Figure 2.2, we associate the variable B to the behavior bound by the ζ operator, denoting the whole behavior definition of the matching actor. In the latter, we rather describes behavior sets using the program point label of the actor that syntactically defined them.

2.1.4 Examples

We now illustrate CAP use considering some examples. These examples show some specific CAP mechanisms and the possible uses of non uniform behaviors.

Figure 2.2 CAP transition relation.

$$\begin{array}{c}
B = [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}] \quad \left\{ \begin{array}{l} m = m_k, \\ \text{length}(\tilde{y}) = \text{length}(\tilde{x}_k), \\ k \in [1, \dots, n] \end{array} \right. \\
\hline
a \triangleright^l B \parallel a \triangleleft^{l'} m(\tilde{y}) \xrightarrow{(l, l_k, l')} C_k[e_k \leftarrow a, s_k \leftarrow B, \tilde{x}_k \leftarrow \tilde{y}] \quad \text{COMM}
\end{array}$$

Figure 2.3 CAP structural context rules.

$$\begin{array}{c}
\frac{D \equiv C \quad C \longrightarrow C' \quad C' \equiv D'}{D \longrightarrow D'} \text{ STRUCT} \quad \frac{C \longrightarrow C'}{C \parallel D \longrightarrow C' \parallel D} \text{ PAR} \\
\frac{C \longrightarrow C'}{\nu x C \longrightarrow \nu x C'} \text{ RES}
\end{array}$$

Figure 2.4 CAP congruence relation.

$$\begin{array}{l}
C \equiv D \quad C \alpha\text{-conv. with } D \quad (\alpha\text{-conversion}) \\
C \parallel \emptyset \equiv C \quad (\text{empty term}) \\
C \parallel D \equiv D \parallel C \quad (\text{commutativity}) \\
(C \parallel D) \parallel E \equiv C \parallel (D \parallel E) \quad (\text{associativity}) \\
T \triangleright T_1 \equiv T \triangleright T_2 \quad \text{if } T_1 \equiv_B T_2 \quad (\text{behavior equivalence}) \\
(\nu a)\emptyset \equiv \emptyset \quad (\text{simplification}) \\
(\nu a)(\nu b)C \equiv (\nu b)(\nu a)C \quad \text{when } a \neq b \quad (\text{permutation}) \\
(\nu a)C \parallel D \equiv (\nu a)(C \parallel D) \quad \text{when } a \notin \mathcal{FN}(D) \quad (\text{scope extrusion}) \\
[m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}] \equiv_B [m_{\pi(i)}(\tilde{x}_{\pi(i)}) = \zeta(e_{\pi(i)}, s_{\pi(i)})C'_{\pi(i)}{}^{i=1 \dots n}] \\
\text{with } \pi \text{ a permutation and } C_i \equiv C'_{\pi(i)} \quad (\text{behavior branch permutation})
\end{array}$$

Figure 2.5 A simple toy example.

$$\begin{aligned}
& \nu a^\alpha, b^\beta, \\
& \quad a \triangleright^1 [m^2(x) = \zeta(e, s)(x \triangleright^3 [n^4(y) = \zeta(e', s')(y \triangleright^5 s)] \parallel x \triangleleft^6 n(e))] \\
& \quad \parallel a \triangleleft^7 m(b) \\
& \quad \parallel a \triangleleft^8 m(b)
\end{aligned}$$

Example 1 In the example described in Figure 2.5, we consider an initial configuration containing an actor, associated to the address a that can receive a message labeled m . In the initial configuration, we can find at program points 7 and 8, two messages of this kind, i.e. sent to the address a and with message label m . When the actor receive one of these two messages, it introduces in the resulting system an actor bound to address b with, as associated behavior, the set of behavior branches defined at program point 3. It also sends in parallel a message labeled n to the address of this new actor. This sent message contains as an argument the address of the initial actor (a). The initial actor on a as well as the matched message m are then removed from the resulting configuration.

After this first transition, we are in a configuration containing a unique actor on address b , as well as two messages: a first one sent to address a and the second one sent to address b . The actor on b can then handle the message $n(a)$ and launch an actor on address a with, as behavior, the initial behavior (defined at program point 1) of the first actor, bound by the initial ζ operator.

We found again the initial state with one message $a \triangleleft m(b)$ less. In that simple example, variables other than s only contain address values. But the initial actor on program point 1 with a particular behavior disappears and reappears two transitions later with its initial behavior.

Example 2 The system described in Figure 2.6 contains two actors. The first one, bound on a , is able to handle two kinds of messages: m and send . The second, bound to b , is only able to handle a single kind of messages, those labeled beh . The initial configuration also contains two messages. A first one is labeled send and targets the address a . The second one is labeled m and targets b .

In the initial configuration, there is only a single computable transition. The actor on a is able to receive the message send . The message m is in the configuration but cannot be handled yet. It is temporarily stuck.

After a first transition between the actor a and the message send , a message labeled beh is sent to the address b ; it contains as argument the set of behaviors of the initial actor on a . The actor on b is able to receive it. In the associated continuation, i.e. in the behavior branch associated to messages labeled beh in this actor, a new actor is installed on address b with as behavior the one received in the message, i.e. in that case the one of the initial actor on a .

Figure 2.6 A simple behavior passing.

$$\begin{aligned}
\forall a^\alpha, b^\beta, \quad & a \triangleright^1 [m^2() = \zeta(e, s)(a \triangleright^3 s), \\
& \quad \text{send}^4(x) = \zeta(e, s)(x \triangleleft^5 \text{beh}(s))] \\
\parallel & a \triangleleft^6 \text{send}(b) \\
\parallel & b \triangleright^7 [\text{beh}^8(x) = \zeta(e, s)(e \triangleright^9 x)] \\
\parallel & b \triangleleft^{10} m()
\end{aligned}$$

Figure 2.7 Non trivial non linearity.

$$\begin{aligned}
\forall a^\alpha, b^\beta, \quad & a \triangleright^1 [m^2(y) = \zeta(e, s)(e \triangleright^3 s \parallel y \triangleleft^4 \text{actor}(s, e))] \\
\parallel & b \triangleright^5 [\text{actor}^6(\text{self}, \text{ego}) = \zeta(e, s)(e \triangleright^7 s \parallel \text{ego} \triangleright^8 \text{self})] \\
\parallel & a \triangleleft^9 m(b)
\end{aligned}$$

This new actor on b is now able to receive the initial message labeled m . This last transition consumes both the interacting actor and the message and re-introduces in the resulting configuration an actor on a similar to the initial one.

This example illustrates the behavior passing mechanism between two actors.

Example 3 The system given in Figure 2.7 is non linear (cf. Chapter 6), but this property is broken here in a non trivial way. Linearity considers addresses as resources and do not accept terms where an address is used twice to install an actor. Even if the initial term is linear, its possible evolutions could break the property.

The actor bound to address a receives the message labeled m . When doing so, it replicates ($e \triangleright s$) and sends its set of behavior branches as well as its name to the address argument of the message m .

The second actor, bound to b , when receiving this new message, replicates and produces a new actor using the parameter of the message.

In that example, after two transitions, there is, in the resulting configurations, two actors bound to a with the same set of behaviors as defined at program point 1.

We recall that the behavior associated to the actor on a with the shape $\zeta(e, s) = (e \triangleright s \parallel C)$ acts as a guarded replication. The actor replicates while C is inserted in the resulting configuration.

Example 4 The linear cell can be modeled in CAP as presented in Figure 2.8. The actor bound to a has two states, i.e. two behavior sets. A first one denotes the empty cell waiting for a put message to receive its value. The second one represents the filled cell waiting for a get message to empty itself and send back the rep message to the get

Figure 2.8 Linear cell.

$$\begin{aligned} & \nu a^\alpha, \\ & a \triangleright^1 [\text{put}^2(v) = \zeta(e, s_{\text{empty}})(\\ & \quad e \triangleright^3 [\text{get}^4(c) = \zeta(e', s_{\text{full}})(c \triangleleft^5 \text{rep}(v) \parallel e' \triangleright^6 s_{\text{empty}})] \\ & \quad]] \end{aligned}$$

message argument. Initially it is associated to the empty state. Such a term has to be put in a context to be reduced.

A static analysis must be able to represent such cyclic behaviors while catching the dependency relation between the value introduced in the cell and the one sent back to the argument of the get message.

Example 5 Finally the last example proposed here is our more high level one, the replicating server example described in Figure 2.9.

Initially the system contains a server on address a and a replicating server on address b . When the server receives the message reify , it sends its address and behavior to the replicating server and dies.

The replicating server introduces two new addresses and associates them to the behavior of the preceding server on a . It also introduces a new actor on a . For each message received, this *Aux* behavior propagates the message to the replicated servers and produces a new actor on address f to receive their response. The first response is transmitted to the initial target and the second one destroyed.

In such a system, we are interested in statically determining which actor is able to receive the different messages sent as well as checking that no message remains forever pending in the configuration.

2.2 STATIC ANALYSIS METHODS

We now present the different methodologies of static analysis. We give their pros and cons as well as the necessary steps in their sound definition.

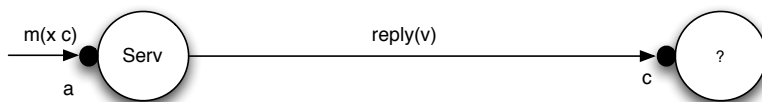
2.2.1 Type system-based

The typing of programming languages has been the subject of many works. It has been particularly applied with success in the context of functional or object-oriented programming. The principal idea is to associate to each sub-term a *type* denoting a collection of potential values. A type system is defined by a type definition or a set of type definitions as well as typing rules. These rules express the condition that sub-terms of a given term must fulfill in order to

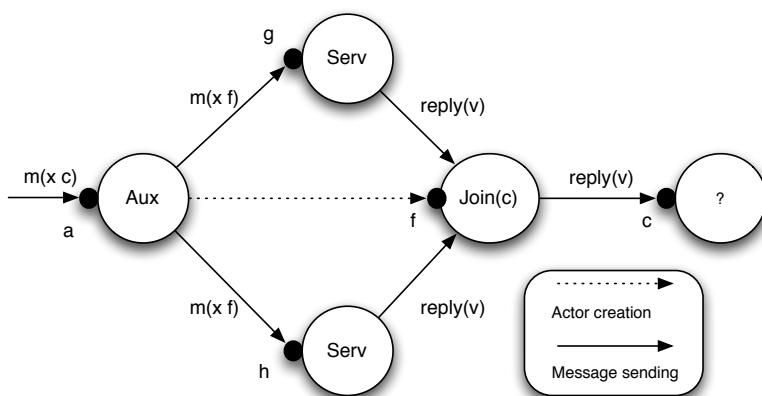
Figure 2.9 Replicating server.
$$\begin{aligned}
\text{Serv} &= [\text{m}^5(\tilde{x}, c) = \zeta(e, s)(e \triangleright^6 s \parallel \dots \parallel c \triangleleft^7 \text{reply}(\tilde{x})), \\
&\quad \text{reify}^8(c) = \zeta(e, s)(c \triangleleft^9 \text{state_Serv}(s, e)) \\
&] \\
\text{DupServ} &= [\text{state_Serv}^{10}(\text{self}, \text{ego}) = \zeta(e, s), \nu g^\delta, h^\theta \\
&\quad (g \triangleright^{11} \text{self} \parallel h \triangleright^{12} \text{self} \parallel \text{ego} \triangleright^{13} \text{Aux}(g, h) \parallel e \triangleright^{14} s) \\
&] \\
\text{Aux}(g, h) &= [\text{m}^{15}(\tilde{x}, c) = \zeta(e, s), \nu f^\iota \\
&\quad (g \triangleleft^{16} \text{m}(\tilde{x}, f) \parallel h \triangleleft^{17} \text{m}(\tilde{x}, f) \parallel f \triangleright^{18} \text{Join}(c) \parallel e \triangleright^{19} s), \\
&\quad \text{reify}^{20}(c) = \zeta(e, s) \\
&\quad (c \triangleleft^{21} \text{state_Serv}(s, e) \parallel a \triangleleft^{22} \text{reify}(c) \parallel b \triangleleft^{23} \text{reify}(c)) \\
&] \\
\text{Join}(c) &= [\text{reply}^{24}(\tilde{y}) = \zeta(e, s)(c \triangleleft^{25} \text{reply}(\tilde{y}) \parallel e \triangleright^{26} [\text{reply}^{27}(\tilde{y}) = \emptyset])] \\
\end{aligned}$$

$$\nu a^\alpha, b^\beta, c^\gamma, a \triangleright^1 \text{Serv} \parallel a \triangleleft^2 \text{reify}(b) \parallel a \triangleleft^3 \text{m}(\tilde{x}, c) \parallel b \triangleright^4 \text{DupServ}$$
Figure 2.10 Replicated server use.

(a) Server use before replication



(b) Replicated server



allow its typing. Types usually contain behavioral information and well-typed terms, depending upon the nature of type information, prevent bad behavior from occurring during execution.

A type definition involves a precise characterization of type values and the proof of two properties: subject reduction and type validity. The first one states that each computation preserves the type computed. It is called *continuity*. Types associated to a sub-term must be invariant with respect to the semantics of the analyzed system. The second one ensures that a typed program does not produce semantic errors.

A type system can then be used in two flavors. The first one is the *type inference* approach. An analyzed system is given as is, without any annotation. The typing algorithm then takes the system description and associates a type to each sub-term, until obtaining a type for the whole system. The second approach is *type checking*. In that case, the system description is provided with type values associated to some or all sub-terms. The typing algorithm ensures that the provided types are consistent with respect to the typing rules.

This approach to static analysis has many advantages. It is syntax-directed, associating types to sub-terms. In that sense it can be natural to define a type system for a program. Depending on the approach chosen: inference vs. checking, the type information could be more or less complex. In the first case type systems usually infer weaker properties than the one that are provided in the second case. But this first approach does not necessitate annotation and can be applied to already developed programs.

Another good point of type-based analysis is the capability to compose terms. As the type algorithm is applied locally on the term, assigning or checking a type to each sub-term, defining a new system by composing terms does not complicate the analysis. A new type is just computed considering the already available types of each sub-system.

However type systems are hardly able to represent very high level properties. They lack a precise handling of control flow and often fail to differentiate non uniform properties depending on recursive instances. Furthermore their extension to analyze another related concern often requires to prove both subject reduction and type validity from scratch, discarding previous efforts.

2.2.2 Model checking

Another famous static analysis technique which is widely used is model checking. Model checking is the process of checking whether a given structure is a model of a given logical formula.

Model checking is also a natural approach to the static analysis. It requires to represent in a structure, usually a graph representation, all the possible behaviors of a system. The logical formula, either a safety property, expressed as an

invariant, or a liveness one, relying on more complex paths along the graph, is then checked within the representation of the considered system.

The principal problem of this approach is the representation of a system in the structure. Big systems or infinite ones can hardly be representable as finite computer-data graph structures. However the exhaustive representation of a program possible behaviors gives access to very powerful analysis. For example, it is one of the only techniques able to handle well in practice these liveness properties.

2.2.3 *Abstract interpretation*

The idea behind the framework of Abstract Interpretation [34] is to rely on well-known results of discrete mathematics to describe and compute properties of programs, derived from their execution semantics. The mathematical bases come, on the one hand, from structural properties of complete lattices or complete-partial orders, and their preservation by monotonic mappings or semi-dual Galois connections, and, on the other hand, from fixed point theorems, like those by TARSKI [103], and their constructive versions by KLEENE.

A software system under analysis has first to be expressed within this framework. A famous dogma of abstract interpretation is that any semantics can be formalized as fixed points of monotonic operators on a complete lattice. Once the original concrete system is expressed in this way, it is related to an abstract lattice, via a pair of functions known as the Galois connection. The semantics can then be computed in the abstract lattice using the original concrete semantics and the Galois connection. This approach allows to construct an abstract over-approximation of the properties satisfied by the concrete system.

Many interesting problems in computer science and in software are, in general, undecidable; this means that they cannot be answered in all cases by any computer program devised for that purpose. The Abstract Interpretation builds sound and decidable but necessarily incomplete analyses. Decidability means that we obtain a result in finite time. Soundness means that the result actually denotes an over-approximation. The drawback lies in incompleteness which expresses that, while absence of errors can be fully trusted, there may be reported errors that are “false positives”. This is often the case when the abstract lattice does not allow a sufficiently precise model of the considered property.

Relating abstract and concrete properties

A concrete semantics is then defined on a lattice $(S, \sqsubseteq, \perp, \cup, \top, \cap)$ using a fixed point definition relying on a monotonic map $F : S \rightarrow S$. We often call *concrete properties* the elements of this concrete lattice.

A second lattice, denoting the abstract representation of concrete properties is introduced $(S^\#, \sqsubseteq^\#, \perp^\#, \cup^\#, \top^\#, \cap^\#)$.

The general idea is to compute an over-approximation of the concrete semantics using the abstract lattice. For example we can consider a least fixed point formulation of the semantics $\text{lfp}_{\perp} F$ or a greatest fixed point one $\text{gfp}_{\top} F$.

In fact, relying on TARSKI fixed point theorem, the least fixed point exists and is defined as the least upper bound of the iterates. However this computation is generally transfinite.

Therefore we relate elements of S to elements of $S^{\#}$. The fixed point is then computed in the abstract lattice and projected back into the concrete one.

The construction of abstract interpretation imposes

1. to express semantics using monotonic operators, in order to be able to apply TARSKI fixed point theorem or its constructive version;
2. to relate elements from the concrete and abstract lattices in a sound manner.

GALOIS CONNECTION BASED ABSTRACT INTERPRETATIONS The initial proposal and the most well-known is the semi-dual Galois connection-based abstract interpretation as presented in [34, 35]. A Galois connection between two lattices is defined as a pair of monotonic functions (α, γ) , preserving the pre-order properties as follows:

Definition 2.1 (Galois connection) *Let (S, \subseteq^S) and $(S', \subseteq^{S'})$ be two partially order sets (posets). We introduce a Galois connection between these two posets as a pair of functions (α, γ) such that:*

- $\alpha : S \rightarrow S'$ (abstraction);
- $\gamma : S' \rightarrow S$ (concretization);
- $\forall x \in S, x' \in S', \alpha(x) \subseteq^{S'} x' \Leftrightarrow x \subseteq^S \gamma(x')$.

Therefore, we can deduce as in [32] that $x \subseteq \gamma(\alpha(x))$. And using a monotonic operator F : $F(x) \subseteq^S F(\gamma(\alpha(x)))$. We obtain:

$$\text{lfp}_{\perp} F(x) \subseteq \text{lfp}_{\perp} F(\gamma(\alpha(x))) \quad \text{gfp}_{\top} F(x) \subseteq \text{gfp}_{\top} F(\gamma(\alpha(x)))$$

This allows to consider Galois connection-based abstract interpretation as sound-by-construction: the monotonic properties of both the concrete semantics operators and the Galois connection functions allow to transfer the fixed point computation from the abstract to the concrete lattice defining sound post-fixed point values in the concrete lattice.

CONCRETIZATION-BASED ABSTRACT INTERPRETATIONS However this kind of Galois connection-based abstractions is not always easy to define as it imposes strong requirements on the existence of the pair of functions. In the preceding case, both functions, abstraction and concretization, were used at each step of the fixed point computation. But it may happen that at least one of them is not computer representable.

The relaxed framework of [36] solves this problem. It introduces abstractions that are either concretization-based or abstraction-based. In this current thesis, we rely on the concretization-based abstraction of the semantics.

In that view, an abstract semantics is defined by a pre-ordered set $(S^\#, \sqsubseteq)$, an abstract iteration basis $\perp^\#$, a monotonic concretization function $\gamma : S^\# \rightarrow S$ and a monotonic abstract semantics function $F^\#$.

The abstract semantics functions must be proved sound with respect to their associated concrete functions. Let $f^\#$ be an abstract operator and f its related concrete function. They must satisfy the following property:

$$\forall x^\# \in S^\#, (f \circ \gamma)(x^\#) \sqsubseteq^\# (\gamma \circ f^\#)(x^\#)$$

WIDENING AND NARROWING TECHNIQUES An important feature of the abstract interpretation methodology is to provide widening and narrowing operators that allow to accelerate the convergence of the post-fixed point computation. These operators are sound in the sense that they over-approximate the real concrete properties but they may be too wide and weaken the precision of obtained results.

In practice, we use such widening operators when computing fixed points in lattices that admit infinite chains with respect to their partial order. In that case, a finite iteration is not able to reach a post-fixed point in general. Using widening operators solves the problem, guaranteeing to reach a post-fixed point in a finite number of iterations.

Abstracting properties: abstract domains

We denote by *abstract domains*, lattices or posets associated to a set of abstract operators. These domains can be combined to produce new ones, preserving the necessary soundness properties. We first present some domain combinations and mention existing abstract domains.

CARTESIAN PRODUCT Building Cartesian products of abstractions is a widely used technique. Let \mathcal{A} and \mathcal{B} be two abstract domains build on the sets $C_1^\#$ and $C_2^\#$ respectively. The obtained domain is build on pairs in $C_1^\# \times C_2^\#$. Pre-order function, union, and widening are defined pair-wise. The γ function is defined as the meet of respective concretization functions:

$$\gamma((c_1, c_2)) = \gamma_A(c_1) \cap^S \gamma_B(c_2)$$

DOMAIN REDUCTION Another great technique is the domain reduction. A reduced domain is produced by exhibiting a reduction operator ρ applied on all elements of the domain. Each abstract computation relies on it to obtain the final abstraction. It must satisfy the soundness assumption:

$$\forall a \in \mathcal{C}^\#, \gamma(a) \subseteq^S \gamma(\rho(a))$$

Existing domains

Since the beginning of abstract interpretation in 1977, many abstract domains have been introduced in the literature. Furthermore existing analyses have been rephrased as abstract interpretation domains, allowing them to be combined or used in other contexts.

Among existing domains, we can mention the great number of domains addressing the abstraction of numerical properties: intervals [33], affine relationships between variables [68], polyhedra [37], octagons [83], and abstract domains devoted to the sound over-approximation of floating point number arithmetic [59].

Another set of domains were introduced to define shape analysis. They allow to represent alias properties [45], pointers and strings manipulations [7] or heap properties using separation logic [102].

Composition of abstract interpretations

In this very general framework of abstract interpretation, every semantics could be potentially expressed as a least fixed point or a greatest fixed point and abstracted. However, its practical use often rely on a least fixed point formulation of the collecting semantics of the system or its trace semantics.

Compared to the type system approach, the composition of these abstractions is not directly obtained in this framework, as it requires to compute a fixed point considering the already computed local abstract fixed points.

The practical approach of [80] for the development of abstraction for analyzing Java program addresses this problem. It targets to provide an abstraction for classes that can be used whatever their call context.

In that view, the semantics of class specific abstraction composition is defined as a greatest fixed point built over the collecting trace semantics of each class, itself expressed in a usual way, as a least fixed point.

This approach allows to provide abstractions that could be directly composed without a great number of iterations to be computed.

This methodology of Abstract Interpretation is, in the author opinion, the most promising. It provides a powerful approach that relies on strong existing results of discrete mathematics. It copes well with the undecidability results of

most computer-related problems using its incomplete approach to static analysis, *i.e.* over-approximating fixed point of a concrete semantics. These over-approximation techniques give also the soundness guarantees that allows to deal with big or even infinite systems, providing decidable abstractions.

2.2.4 Flow logic

Finally a last approach is the Flow logic methodology [92]. It considers an abstract representation of control flow information and is a technique in-between type system approach and abstract interpretation.

A Flow logic analysis is defined either in a syntax directed way, using inductive methods or in a more abstract way, based on co-inductive methods. Abstract properties are then combined to obtain sound wider ones.

This approach to static analysis based on a control flow or constraint-based view of a system has to be related with HM(X) type systems [94]. Such type systems are built over a set of constraints X . When typing a term, the typing algorithm produces a set of constraints. The term is well-typed iff the produced constraints are satisfiable. Solving constraints may require complex algorithm like [5, 6] and even widening techniques.

2.3 CONCURRENCY ANALYSIS RELATED WORKS

2.3.1 Typing process calculi

Initial types for concurrency

Type systems have been largely applied on process algebras. A first *sort discipline* was proposed for the polyadic π -calculus. It associates to each channel its arity as well as the arity of the communicated channels. An inference algorithm was proposed by Gay in [57]. PIERCE and SANGIORGI have then extended these sorts in [96] to describe the usage of channels, allowing to characterize names used to read, to write or both.

More complex typing: type checking based

In the context of concurrent objects, NIERSTRASZ introduced in [93] a type checking algorithm to specify the communication protocol. NAJM and NIMOUR [46, 85, 86, 87] apply this technique to objects changing dynamically of interface.

Other works involving complex type expressions like session types were developed. They rely on a checking approach to typing. The user provides a specification of the protocol to be checked or he helps the checking algorithm with partial type information.

Among these works, we can cite BOUDOL and AMADIO [10, 18] for π extensions, RAVARA, TOKORO and VASCONCELOS [99, 104] for a concurrent object calculus named TyCo, PUNTIGAM [97] for another concurrent objects framework, DAL ZILIO [42] for the blue calculus, and KOBAYASHI and YONEZAWA [71, 75, 76] concerning lock-freedom or deadlock-properties for the π calculus.

Inference based approaches for high level properties

Concerning CAP, the identified properties of interest, among traditional usage properties or control flow analysis, are the linearity property and the orphan-freeness property. The first one sees addresses as resources and ensures that no address is used twice to bound an actor in a reachable configuration. The second one targets to ensure that all sent messages will be handled in the future and are not stuck waiting for an actor to handle them. These two properties are complex to be checked due to both their description, the non-uniform interfaces of CAP and its behavior passing ability.

The analyses proposed in [25, 26, 27, 28, 29] address these problems using a HM(X) based type inference algorithm. In that context, constraints generated when typing a term describe the causality dependencies among the different actor interfaces. They rely on sub-typing constraints to express such relations. Similarly to the previously mentioned type systems, their type information denotes the behavior of the system, but they are automatically inferred.

Another interesting work concerning the analysis of high-level properties in a type inference flavor is the recent work of KOBAYASHI [70] that expresses deadlock properties using complex extensions of its previous type systems but providing an inference algorithm.

2.3.2 *Flow logic analysis of concurrency*

The Flow logic framework introduced by NIELSON and NIELSON in [92] has been applied to different process algebras like the π -calculus in [15, 16], Ambients [90] or spi with applications to security in [17, 88].

However, similarly to the type system approach it hardly deals with non-uniform properties.

2.3.3 *Model checking and concurrency*

Few works address the use of model checking in order to validate concurrent communicating systems. One can cite two different approaches. A first one as used by CRIDLIG [38], HUTCH [63], REHOF [12] or MONTANARI and PISTORE [84] aims at expressing a process algebra or a concurrent language into a finite representation in a target model such as Petri nets, LOTOS or any inherently concurrent model fitted with model checking tools.

A second one, addressed by DELZANNO [44], AMADIO [8] or BRUNI [20], do not target an effective verification of system but rather to define theoretical result on the decidability of strong properties like the representation of the reachability set. They usually consider more general structures than just process algebras, called nominal calculi.

2.3.4 *Abstract interpretation-based analysis of process calculi*

Early works address the use of abstract interpretation in a concurrent context. In [65], JAGANNATHAN and WEEKS rely on it to analyze a parallel language using shared variables. In [11], ANDREOLI, CASTAGNETTI and PARESCHI compute linear logic formulas describing evolutions of resources for a concurrent language.

SCHMIDT introduces in [101] the first abstract interpretation-based analysis, describing communication, for the π -calculus. In [105, 107] VENET gives a very powerful approach to the analysis of the π -calculus. It relies on ideas of [66] to make explicit the control flow and using an alias-like analysis as in [106] provides an abstraction allowing to distinguish between recursive instances of threads. However his analyses necessitates the π term to be in a special form allowing it to be analyzed, in particular they were not able to deal with nested guarded replications.

Later, FERET has extended this framework and applied it to the usual polyadic π with guarded replications [50]. He also handles other paradigms of concurrency such as Ambient in [51] or spi calculus. In [49], he generalizes this approach and proposes a generic framework for the analysis of mobile systems. Calculi are first expressed in a non standard form, like the one of Venet, allowing to distinguish recursive instances of threads and values. Then relational abstractions allow to over-approximate non-uniform properties of control-flow and to count occurrences of threads in reachable configurations.

The current thesis follows this line of thoughts.

2.4 OVERVIEW OF CONTRIBUTIONS

In this thesis we introduce an abstract interpretation-based static analysis of CAP. The analyses presented are devoted to CAP but this general approach can be easily generalized to any other calculus or more high level description. As presented in the conclusion chapter, Chapter 9, the studied properties could be applied to other calculi, like the π -calculus in order to verify strong properties. Similarly this framework of abstract interpretation of concurrent process algebra eases the extension of the analyses and could be extended to handle other properties as well.

This document is structured as follows.

2.4.1 *CAP non standard semantics*

The Chapter 3 introduces the abstract interpretation framework for the analysis of concurrent systems. It gives the encoding of CAP in its non standard form that facilitates its latter abstraction and analysis. Such non standard encoding expresses configuration as set of threads, making explicit the history of transitions that leads to the creation of both values and threads. Each thread is then defined by a program point in the considered term, a identity marker and an environment assigning values to variables.

The non standard encoding of CAP is proved strongly bisimilar to the standard CAP semantics presented in this chapter.

2.4.2 *Abstracting collecting semantics and abstract domains*

The same Chapter 3 gives an insight to the generic abstract domains presented in [49] for abstracting the collecting semantics of terms. It presents the two principal existing generic abstract domains: the control flow abstract domain, associating an abstract environment to each thread program point, and the occurrence counting abstract domain, that represents numerical properties of thread occurrences in reachable configurations.

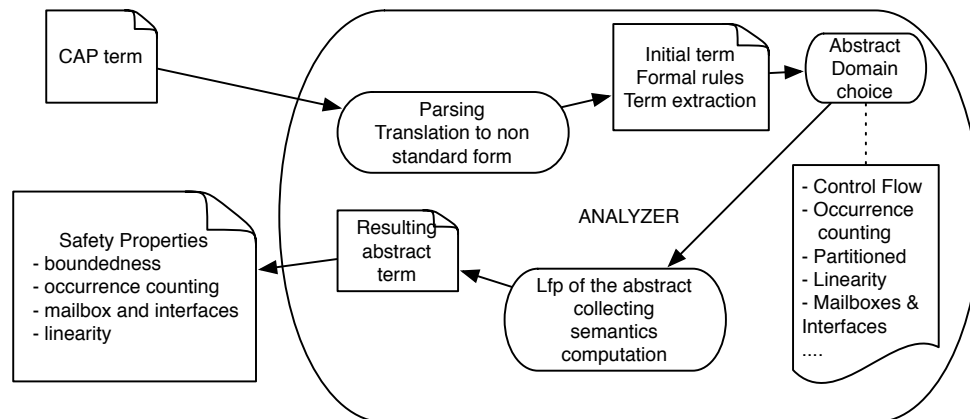
The Chapter 4 introduces a new abstraction which is central in our CAP specific analysis: the partitioned abstract domain. This domain, which is parametric, allows to represent properties verified by the subset of threads associated to each address. For a given address a , it gives the properties of actors installed on this address and messages sent to it using the underlying abstract domain.

The Chapter 5 proposes three enhancements concerning the original occurrence counting abstraction. These extensions avoid some spurious transitions that were previously computed and give a much better precision in the overall abstract element obtained.

2.4.3 *Linearity*

The Chapter 6 proposes a static analysis in this framework for the linearity property. This property expresses that each address is associated to at most one actor. It sees address as exclusive resources that cannot be shared among multiple actors at the same time.

This analysis is performed using a specific abstract domain that computes a usage value for each binder and thread variables.

Figure 2.11 Framework overview.

2.4.4 Orphan freeness checking

The Chapter 7 presents an analysis ensuring the orphan freeness property. In such an asynchronous context, orphans are defined as messages that are sent to an address where no actor is and will ever be able to handle them.

The analysis is performed in two steps. The first one relies on our abstract interpretation framework and computes an over-approximation of the possible sequences of actor behaviors associated to each address. It also gives an approximation of messages available at each of these steps. This abstraction is computed relying on the above partitioned abstract domain to obtain an address-partitioned representation. Then the second one verifies the property in each abstract element obtained.

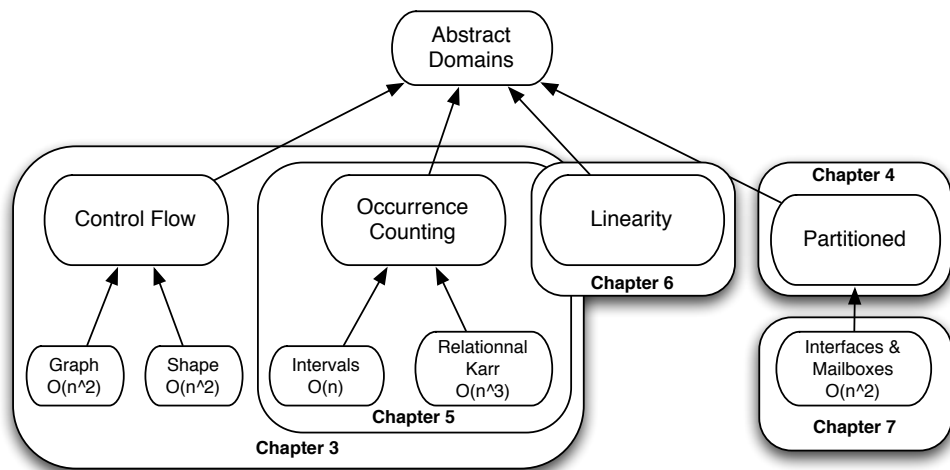
2.4.5 Implementation issues

Finally the last contribution is presented in the Chapter 8. It is a tool implementing all the abstract domains presented. It performs all the steps mentioned above and presented in Figure 2.11. It takes a CAP term, transform it in its non standard encoding and computes the least fixed point of its collecting semantics using a reduced product of the abstract domains presented here.

The Chapter 9 concludes and presents the perspectives opened by this work.

The Figure 2.11 gives an overview of the proposed analysis framework and the Figure 2.12 illustrates the abstract domains used and the chapters where they are introduced.

Figure 2.12 Hierarchy of abstract domains.



This chapter addresses the first essential step of our analysis: it defines the expression of CAP into FERET's framework [49].

The initial breakthrough of the static analysis of mobile systems using abstract interpretation in the line of works of VENET and FERET, is the explicit representation of history of transitions that led to the creation of values and processes. These history markers allow to disambiguate recursive instances of the same binder or of the same syntactic process definition.

A first step in this approach is therefore the expression of our calculus, CAP, in this system, explicitly representing such history markers. We call this encoded semantics the non standard semantics.

Later, the analyzed term is encoded in its non standard form and its collecting semantics abstracted. Approximating markers gives relevant information on threads and overpasses 0-CFA analysis which does not consider such information and is not able to differentiate such processes or values.

The chapter describes the encoding of CAP into the non standard semantics as presented by FERET in [49]. The first Section 3.1 gives the intuition behind both FERET's framework and CAP encoding in it. The Section 3.2 details the framework instantiation to model CAP and its specific features. Then the Section 3.3 presents the principal abstract domains defined by FERET in [49]: the control flow abstract domain and the occurrence counting abstract domain. They are presented here as a basis of our own framework of abstract interpretation for CAP terms. Finally we conclude the chapter in Section 3.4.

This work has been firstly defined in [53] but in an incomplete version. This chapter rather extends the later approach of the encoding as presented in [55] and developed in [56].

3.1 THE INTUITION

3.1.1 FERET's *framework*

We now briefly present the main ideas behind the current approach of concurrent systems analysis. We first give the intuition that motivates the proposal of FERET's generic non standard semantics and then explain how to express specifics of CAP in it.

Non standard semantics encoding

PRINCIPAL IDEAS Usual semantics of concurrent calculi are defined using the now traditional MILNER approach. The semantics is then described using transition rules, a congruence relation and some compatibility conditions. A drawback of static analyses that directly rely on this kind of semantic description is the ambiguity that comes from α -conversion. In the congruence relation, the standard α -conversion rule allows to rename recursive instances of the same binder with fresh names in order to differentiate them. However existing static analyses, like 0-CFA, merge the information associated to these recursive instances leading to imprecise results.

The approach proposed by VENET and later FERET, that relies on abstract interpretation to over-approximate the collecting semantics of terms, necessitates to make the control flow information of terms explicit. Therefore they introduced a notion of non standard semantics, bisimilar to the usual standard one, that is enriched with control flow data.

Each value is then described by a pair: the binder and a history marker. The history marker is a tree denoting the sequence of transitions that led to the creation of this particular value. In the following, we denote by \mathcal{M} such set of markers.

In order, to differentiate recursive instances of the same process, as well, each process is also tagged by a history marker denoting the sequence of transitions that led to its creation.

STATE In the non standard encoding, a state of the system is represented as a set of threads. Each thread is a triple (p, id, E) composed of a program point p , a history marker id of the process and an environment E associated to the thread.

The environment associates a pair composed of a program point and a history marker to variable names.

Program points are automatically attributed to the interesting parts of the analyzed term. They denote sub-terms that become threads as well as binders creating values.

SEMANTICS The current operational semantics is not based on a labeled transition relation and a congruence relation but rather on a set of reduction rules. The operational semantics is guided by a set of formal rules.

When a transition is computed on a given state according to one of the defined formal rules, it considers one or more interacting threads. The formal rule describes the number of interacting threads, the necessary conditions for the transition to occur, and the value passing induced by the transition. These conditions as well as the value passing rely on environment data.

Then the resulting state is computed by removing some of the interacting threads as well as launching new ones. The threads to be launched are identified according to the interacting thread program points.

SOUNDNESS In practice, we do not represent markers as trees built over transition labels. Abstracting them to words of program points is enough to differentiate recursive instances of both values and processes. In that case we only record program points associated to the sub-term that denotes recursivity. We have then $\mathcal{M} \triangleq \mathcal{L}_p^*$.

An encoding is sound when created markers respect this condition of freshness: two recursive instances cannot be bound to the same marker. If the formal rules respect some conditions in their definition, this is obtained by construction.

Another necessary condition is the bisimulation proof between the standard semantics and its non standard encoding.

3.1.2 Instantiating the framework to model CAP semantics

In [49], FERET introduces a set of encodings into the non standard semantics. He gives an encoding for the π calculus, the spi, the ambient and bio-ambients. All these calculi have their own specific features but are synchronous and first order. We now give the details of CAP encoding.

CAP features

ASYNCHRONICITY CAP is an asynchronous calculus. A way to encode this is to associate program points, and therefore threads, to communications, *i.e.* message sendings. These program points are not associated to any continuation. A transition describes an actor handling a message. In that view, only actors define continuations, *i.e.* threads to be launched.

HIGH-ORDERNESS One of the main difficulties is to model high-orderness in the non standard semantics. We choose to model it using persistent definitions. Some threads then denote behaviors. This choice pollutes states with threads that may not be used anymore but it eases the representation of high-order and its later abstraction.

Like in the Join calculus, computing transitions sometimes produces behavior threads. When launching an actor with a statically defined behavior, we produce one behavior thread for each behavior branch of its behavior set. The ζ operator in behavior branches allows then to bind the behavior thread to a variable that may be used later or be sent inside messages.

NON UNIFORM INTERFACES Non uniform interfaces is the notion for actors to accept at a moment some set of message labels and later another one. In our encoding, actors on the same address are represented by different threads depending on the sub-terms that defined them. We identify the address of an actor using the thread environment. The interface is characterized by either the program point of the thread in case of a syntactically defined actor or the thread environment in case of an actor which behavior is defined by a variable, *i.e.* a dynamic actor.

3.2 INSTANTIATING THE GENERIC FRAMEWORK

3.2.1 *Generic framework semantics*

The generic non standard semantics is very parametric. We now present the different parts that have to be defined in order to encode a new calculus.

Partial interactions

Partial interactions are associated to active sub-terms of the analyzed term. Each sub-term labeled by a program point is able to exhibit a set of partial interactions. These partial interactions define the variables of the matched sub-terms that are parameters of the threads and the ones that are bound by value passing. It also defines the continuation associated to the program point.

The partial interaction specifies whether the thread is consumed or not when interacting.

Formal rules

Formal rules are tuples that drive transitions. They describe the number of interacting threads, the partial interactions that each interacting thread program point must be able to exhibit, the constraints between thread parameters that allow the transition computation, and the value passing produced by the transition computation.

Syntax extraction

The term is automatically labeled with program points. The syntax extraction associates a partial interaction to each labeled sub-term. It also defines the initial term.

3.2.2 *Partial interactions*

Here, in CAP, partial interactions can represent a syntactically defined actor, one of its particular behavior branch, a dynamic actor (an actor whose behavior

is defined by a variable) or a sent message. This encoding, differentiating syntactic actors from dynamic ones, allows us to deal with behavior passing. Once a behavior has been declared, it is present in the configuration as a behavior thread, while a reference to it is used in messages. Dynamic actors could then use such a behavior if they are associated to the corresponding reference.

We thus define the set of partial interaction names

$$\mathcal{A} = \{\text{static_actor}_n, \text{behavior}_n, \text{message}_n \mid n \in \mathbb{N}\} \cup \{\text{dynamic_actor}\}$$

and their arities as follows:

$$\text{Ari} = \left\{ \begin{array}{l} \text{static_actor}_n \mapsto (2, n + 2), \\ \text{behavior}_n \mapsto (1, n + 2), \\ \text{dynamic_actor} \mapsto (2, 0), \\ \text{message}_n \mapsto (n + 2, 0) \end{array} \right\}$$

Partial interaction arities define the number of parameters and the number of bound variables.

Both partial interactions static_actor_n and behavior_n denote a particular behavior of an actor. The first one denotes the set of behavior branches syntactically defined at that point in the term when the second characterizes one of the behavior branch of a behavior set definition. The first one is associated to an address when the second one is alone and can be used with a dynamic actor. The second one acts as a definition and stays in the configuration when used, whereas the first one is deleted. They are parametrized by their message label and bind $n + 2$ variables, the two variables under the ζ operator expressing reflexivity as well as the parameters of the message they can handle. The first one is also parametrized by its actor address.

The partial interaction dynamic_actor denotes a thread representing an actor. It is consumed when interacting. It only has two parameters: its name and set of behaviors. It binds no variables.

Finally the partial interaction message_n represents the message that is sent to a particular address (actor). So it has $n + 2$ parameters: one for the address, one for the message name and n for the variables of this message. It is consumed when interacting.

In the framework, we also have to define a type associated to each partial interaction. The behavior_n partial interaction receives the type replication that means that it will not be removed from configurations when interacting. All other partial interactions are associated to the type computation, which denotes their consumption when interacting.

3.2.3 Formal rules

We now define the formal rules that drive the interaction between threads. A transition computation relies on one of the defined rules. Each rule is defined as a tuple that denotes

1. the number of interacting threads;
2. the partial interaction that interacting threads must exhibit;
3. the compatibility constraints among interacting thread environments;
4. finally the value passing induced by the transition computation.

We recall that the partial interaction associated to each interacting thread defines the number of parameters and binds variables respectively used by compatibility constraints and value passing.

In the case of CAP, we have two rules that describe an actor handling a message, depending on the kind of actor we have, a static or a dynamic one.

In the following, the i -th parameter, the j -th bound variable, and the identity of the k -th partial interaction are respectively denoted by X_i^k , Y_j^k and I^k .

Both rules mimic the same schema: compatibility ensures that the message can be received by a behavior of the actor and value passing binds variables of the message in the actor behavior to the message parameters. The value passing also computes the binding for the zeta (ζ) operator variables.

Communication with a static actor

The first rule needs two threads, the first one must denote a partial interaction `static_actor` when the second one must denote a partial interaction `messagen`. We both check that the actor address (X_1^1) is equal to the message receiver (X_1^2) and that the actor behavior name (X_2^1) is equal to the message name (X_2^2).

We then define `v_passing` that describes the value passing due to both the ζ operator and message handling.

The Figure 3.1 defines the `static_transn` formal rule.

Communication with a dynamic actor

The second rule needs three threads: the first one must denote a partial interaction `behaviorn`, the second one a partial interaction `dynamic_actor` and the third one a message `messagen`. We check the equality between actor address (X_1^2) and receiver (X_3^3), behavior name (X_1^1) and message name (X_2^3).

Figure 3.1 Communication with a syntactically defined actor.

$$\text{static_trans}_n = (2, \text{components}, \text{compatibility}, \text{v_passing})$$

where

$$1. \text{ components} = \begin{cases} 1 & \mapsto \text{static_actor}_n, \\ 2 & \mapsto \text{message}_n \end{cases};$$

$$2. \text{ compatibility} = \begin{cases} X_1^1 = X_1^2; \\ X_2^1 = X_2^2; \end{cases};$$

$$3. \text{ v_passing} = \begin{cases} Y_1^1 \leftarrow X_1^1; \\ Y_2^1 \leftarrow I^1; \\ Y_{i+2}^1 \leftarrow X_{i+2}^2, \forall i \in \llbracket 1; n \rrbracket; \end{cases}.$$

BEHAVIOR SET FUNCTION We define the endomorphism `behavior_set` on the set $\mathcal{L}_p \times \mathcal{M}$ as follows: $(p, m) \mapsto (p', m)$ where p is a behavior program point and p' is the program point where p has been syntactically defined.

As an example, in the term $\nu^\alpha a, a \triangleright^1 [\text{foo}^2() = \zeta(e, s)C]$, we have

$$\text{behavior_set}(2, \text{marker}) = (1, \text{marker}).$$

This function is statically defined while parsing the term.

With this `behavior_set` function we check that the behavior value associated to the actor denotes the matched behavior partial interaction.

The value passing is defined in the same way as in the first rule. The Figure 3.2 defines the `dyn_transn` formal rule.

3.2.4 Syntax extraction

We now define the syntax extraction function that takes a CAP term describing the initial state of a system in the standard syntax and extracts its abstract syntax. We first define the mapping interaction from term program points to partial interactions. We then define the β function that is used to build non standard threads from a CAP term.

ASSOCIATING PARTIAL INTERACTIONS TO PROGRAM POINTS We map each program point labeled $l \in \mathcal{L}_p$ to a set of partial interactions it can exhibit and to an interface, denoting its associated thread environment domain.

Figure 3.2 Communication with a dynamic actor.

$$\text{dyn_trans}_n = (3, \text{components}, \text{compatibility}, \text{v_passing})$$

where

$$\begin{aligned} 1. \text{ components} &= \begin{cases} 1 \mapsto \text{behavior}_n, \\ 2 \mapsto \text{dynamic_actor}, \\ 3 \mapsto \text{message}_n \end{cases} \\ 2. \text{ compatibility} &= \begin{cases} X_1^2 = X_1^3; \\ \text{behavior_set}(I^1) = X_2^2; \\ X_1^1 = X_2^3; \end{cases} \\ 3. \text{ v_passing} &= \begin{cases} Y_1^1 \leftarrow X_1^2; \\ Y_2^1 \leftarrow X_2^2; \\ Y_{i+2}^1 \leftarrow X_{i+2}^3, \forall i \in \llbracket 1; n \rrbracket; \end{cases} \end{aligned}$$

A partial interaction pi is given by a tuple

$$\text{pi} = (s, (\text{parameter}_i), (\text{bound}_i), \text{continuation})$$

where $s \in \mathcal{A}$ is a partial interaction name, $(m, n) = \text{Ari}(s)$ its arity, $(\text{parameter}_i) \in \mathcal{V}^m$ its finite sequence of variables (used as X_i variables in formal rules), $(\text{bound}_i) \in \mathcal{V}^n$ its finite sequence of distinct variables (used as Y_i in formal rules), and finally $\text{continuation} \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ its syntactic continuation. Such a continuation is called syntactic because it has to be updated with value passing to be a valid term of the process calculus. We use both sequences (parameter_i) and (bound_i) to compute value passing, finally we deal with the set continuation to determine which threads have to be inserted in the system.

- The label of a program point $a \triangleright^l [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i]^{1 \leq i \leq m}$ is associated to the interface $\{a\} \cup \bigcup_i \mathcal{FN}(C_i)$ and to the following set of partial interactions:

$$\left\{ \begin{array}{l} \left\{ (\text{static_actor}_n, [a, m_1], [e_1, s_1, \tilde{x}_1], \beta(C_1, \emptyset)) \right\} \\ \left\{ (\text{static_actor}_n, [a, m_2], [e_2, s_2, \tilde{x}_2], \beta(C_2, \emptyset)) \right\} \\ \dots \\ \left\{ (\text{static_actor}_n, [a, m_m], [e_m, s_m, \tilde{x}_m], \beta(C_m, \emptyset)) \right\} \end{array} \right\}.$$

Each of the partial interaction in the set denoting one particular behavior branch and therefore a handleable message.

- The label of a program point $\alpha \triangleright^l x$ is associated to the interface $\{\alpha, x\}$ and to the following set of partial interactions:

$$\left\{ (\text{dynamic_actor}, [\alpha, x], [], \emptyset) \right\}.$$

- The label of a program point $\alpha \triangleleft^l m(\tilde{P})$ is associated to the interface $\{\alpha\} \cup \mathcal{FN}(\tilde{P})$ and to the following set of partial interactions:

$$\left\{ (\text{message}_n, [\alpha; m; \tilde{P}], \emptyset, \emptyset) \right\}.$$

- The label of a program point l_i corresponding to a particular behavior of an actor *i.e.* $m_i^{l_i}(\tilde{x}) = \zeta(e_i, s_i)C_i$ is associated to the interface $\mathcal{FN}(C_i) \setminus \{e_i, s_i\}$ and to the following set of partial interactions:

$$\left\{ (\text{behavior}_n, [m_i], [e_i, s_i, \tilde{x}], \beta(C_i, \emptyset)) \right\}.$$

EXTRACTION FUNCTION β Finally, the syntax extraction function β is inductively defined over the standard syntax of the syntactic continuation, as follows:

$$\begin{aligned} \beta((\nu \alpha^\alpha)C, E_s) &= \beta(C, E_s[\alpha \mapsto \alpha]) \\ \beta(\emptyset, E_s) &= \{\emptyset\} \\ \beta(C_1 \parallel C_2, E_s) &= \beta(C_1, E_s) \cup \beta(C_2, E_s) \\ \beta(\alpha \triangleright^l [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i]^{i=1, \dots, n}, E_s) &= \{(l, E_s)\} \cup \bigcup_{i=1, \dots, n} \{(l_i, E_s)\} \\ \beta(\alpha \triangleright^l B, E_s) &= \{(l, E_s)\} \\ \beta(\alpha \triangleleft^l m(\tilde{P}), E_s) &= \{(l, E_s)\} \end{aligned}$$

Remark 1 (Behavior threads) *One could notice that the syntax extraction of a syntactically defined actor generates not only the thread associated to the actor but also all behavior threads defining all behavior branches of the actor. The extraction of a message in which at least one of the variables is a syntactically defined behavior would generate a similar set of behavior threads.*

This mechanism allows to launch behavior threads that may later play their role of available behavior definitions.

Remark 2 (Static environment) *We denote by E_s the environment associated to threads. Threads created by the β function are here pairs of a program point and a static environment. At this point, no marker is present in the thread representation. And the only variables that appears in static environments are addresses created by a ν operator in the local continuation.*

The initial term creation and the transition computation, which relies on the use of formal rules, take these static threads and enrich them with markers:

- a marker denoting thread creation, transforming the thread definition into a triple (pp, id, Env) ;
- a marker denoting value creation (by the ν) is associated to created value.

The initial state for a term \mathcal{S} is described by $init_s$, a set of potential continuations in $\wp(\wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L})))$ defined as $\beta(\mathcal{S}, \emptyset)$.

3.2.5 Operational semantics

We now briefly describe how to use the preceding definitions to express in the non standard syntax both an initial term and the computation of a transition according to one of the aforementioned rules.

Initial configurations are obtained by launching the continuation $init_s$ with an empty marker and an empty environment. That means inserting in an empty configuration one thread for each pair (p, E_s) in $\beta(init_s)$ where each value in E_s is associated to an empty marker.

We focus now on the interaction computation according to one of the two rules defined in Section 3.2.3. First of all, we have to find a correct interaction. It means that we have to find some threads in the current configuration that can be associated to the right partial interactions according to the matching formal rule. Then, we check that their interfaces satisfy the synchronization constraints. Thus we can compute the interaction. All these steps are performed using the generic primitives which are mentioned in parentheses. Their formal definition is given in [49, Chap.4]:

- we remove interacting threads according to the type of their exhibited partial interactions (*exhibits*, *components*);
- we choose a syntactic continuation for each thread;
- we compute dynamic data for each of these continuations (*sync*):
 - we compute the marker (*marker*);
 - we take into account name passing (*vpassing*);
 - we create fresh variables and associate them to the correct values (*launch*);
 - we restrict the environment according to the interface associated to the program point (*launch*);
- we remove interacting threads depending on their types (*remove*).

An explicit definition is given in Figure 3.3.

Figure 3.3 Non standard operational semantics.

Let C be a configuration. Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_passing)$ be a formal rule. Let us be given a tuple $(t^k)_{1 \leq k \leq n} = (p^k, id^k, E^k)_{1 \leq k \leq n}$ of distinct threads and a tuple $(pi^k)_{1 \leq k \leq n} = (s^k, (\text{parameter}_1)^k, (bd_1)^k, \text{continuation}^k)_{1 \leq k \leq n}$ of partial interactions, such that:

1. $\forall k \in \llbracket 1; n \rrbracket, \text{exhibits}(t^k, pi^k)$;
2. $\forall k \in \llbracket 1; n \rrbracket, \text{components}(k) = s^k$;
3. $\text{sync}((t^k), ((\text{parameter}_1)^k), \text{compatibility}) \neq \perp$.

Then

$$C \xrightarrow{(\alpha^i)^n}_{ns} (C \setminus \text{removed}) \cup \text{new_threads}$$

with:

- $\text{removed} = \text{remove}((t^k), (s^k))$;
 - $\text{new_threads} = \bigcup_{1 \leq k \leq n} \text{launch}(Ct^k, \bar{id}, \bar{E}^k)$,
where $\forall k \in \llbracket 1; 3 \rrbracket$:
 - $Ct^k \in \text{continuation}^k$;
 - $\bar{id} = \text{marker}((p^{k'}, id^{k'}, E^{k'})_{1 \leq k' \leq n})$;
 - $\bar{E}^k = v_passing(k, (t^{k'})_{1 \leq k' \leq n}, ((bd_1)^k)_k, ((\text{parameter}_1)^k)_k, v_passing)$.
 - $\forall k \in \llbracket 1; n \rrbracket, \alpha^k = (\mathcal{R}, (t^k), (pi^k))$.
-

3.2.6 Resulting transition system

To illustrate the use of the non standard semantics, we compute the first transition of the Example 2.6, page 27. In order to keep the presentation simple, we do not represent variables denoting message label in thread environments. We admit in the following that the correspondence between message label and behavior branch label is statically determined.

The initial configuration¹ is:

$$\begin{aligned} & (1, \epsilon, \left[\begin{array}{l} \mathbf{a} \mapsto \alpha, \epsilon \end{array} \right]) \quad (2, \epsilon, \left[\begin{array}{l} \mathbf{a} \mapsto \alpha, \epsilon \end{array} \right]) \quad (4, \epsilon, []) \\ & (6, \epsilon, \left[\begin{array}{l} \mathbf{a} \mapsto \alpha, \epsilon \\ \mathbf{b} \mapsto \beta, \epsilon \end{array} \right]) \quad (7, \epsilon, \left[\begin{array}{l} \mathbf{b} \mapsto \beta, \epsilon \end{array} \right]) \quad (8, \epsilon, []) \\ & \quad \quad \quad (10, \epsilon, \left[\begin{array}{l} \mathbf{b} \mapsto \beta, \epsilon \end{array} \right]) \end{aligned}$$

At this point, the only possible transition involves threads on program points 1 and 6 and corresponds to the `static_transn` rule. Program point 1 is able to exhibit the two following partial interactions:

$$\left\{ \begin{array}{l} \left\{ (\text{static_actor}_n, [\mathbf{a}, \text{m}], [e, s], \beta(\mathbf{a} \triangleright^3 s, \emptyset)) \right\}, \\ \left\{ (\text{static_actor}_n, [\mathbf{a}, \text{send}], [e, s, x], \beta(x \triangleleft^5 \text{beh}(s), \emptyset)) \right\} \end{array} \right\}$$

when the program point 6 exhibits the only partial interaction:

$$\left\{ (\text{message}_n, [\mathbf{a}, \text{send}, \mathbf{b}], \emptyset, \emptyset) \right\}$$

We choose the second partial interaction for 1. We first check synchronization constraints. We need that $X_1^1 = X_1^2$ and $X_2^1 = X_2^2$. So $(\alpha, \epsilon) = (\alpha, \epsilon)$ and both threads share the same label `send`. We can now compute value passing, thread launching and removing. We have to remove interacting threads and to add threads in $\beta(\{x \triangleleft^5 \text{beh}(s)\}, \emptyset) = (5, [])$ with their environment updated by value passing. Value passing gives the value of e, s and x , we have respectively

$$(\alpha, \epsilon), (1, \epsilon) \text{ and } (\beta, \epsilon). \text{ Thus the launched thread is } \left(5, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right] \right).$$

We obtain the new configuration:

$$\begin{aligned} & (2, \epsilon, \left[\begin{array}{l} \mathbf{a} \mapsto \alpha, \epsilon \end{array} \right]) \quad (4, \epsilon, []) \quad (5, \epsilon, \left[\begin{array}{l} x \mapsto \beta, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right]) \\ & (7, \epsilon, \left[\begin{array}{l} \mathbf{b} \mapsto \beta, \epsilon \end{array} \right]) \quad (8, \epsilon, []) \quad (10, \epsilon, \left[\begin{array}{l} \mathbf{b} \mapsto \beta, \epsilon \end{array} \right]) \end{aligned}$$

¹ We can notice the absence of threads at program points 3, 5 and 9 which correspond to sub-terms. These are not present in the initial configuration.

Then the only possible computable transition is (7, 5), the actor on address b receiving the new launched beh message. We obtain

$$\begin{array}{l} (2, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \end{array} \right]) \quad (4, \epsilon, []) \quad (8, \epsilon, []) \\ (9, \epsilon, \left[\begin{array}{l} e \mapsto \beta, \epsilon \\ x \mapsto 1, \epsilon \end{array} \right]) \quad (10, \epsilon, \left[\begin{array}{l} b \mapsto \beta, \epsilon \end{array} \right]) \end{array}$$

In this configuration, a new actor is launched, associated to program point 9. This actor is associated to the address (β, ϵ) and its behavior is defined by the variable x bounded to the behavior set $(1, \epsilon)$, *i.e.* $[m^2() = \zeta(e, s)(a \triangleright^3 s), \text{send}^4(x) = \zeta(e, s)(x \triangleleft^5 \text{beh}(s))]$.

Until now all computed transitions involved statically defined actors and the launched thread markers were the one of the interacting actor.

Finally the last transition computable is 9, 2, 10. We obtain

$$\begin{array}{l} (2, \epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \end{array} \right]) \quad (4, \epsilon, []) \quad (8, \epsilon, []) \\ (3, 9.\epsilon, \left[\begin{array}{l} a \mapsto \alpha, \epsilon \\ s \mapsto 1, \epsilon \end{array} \right]) \end{array}$$

In this last transition, involving a dynamic actor, we have the creation of a new marker: $9.\epsilon$.

3.2.7 Soundness

We need to prove the correspondence between CAP semantics and its expression in the meta language. We first state the soundness by construction of marker creation. Then we give the bisimulation result.

Well-formedness conditions

The soundness of marker computation is given by construction of our formal rules. The definition of the generic non standard semantics states (cf. [49, Sect. 4.1.3]) that formal rules involve at most one partial interaction of type replication. Moreover if they involve one partial interaction of type replication they must also involve one partial interaction of type computation. It allows to ensure a thread be consumed when creating a new marker and to represent its identity in the generated marker.

Such well-formedness condition ensures that a thread (p, id, E) can be fully defined by the pair (p, id) .

In other words: let (p, id, E) and (p', id', E') be two threads. Then

$$(p, \text{id}) = (p', \text{id}') \implies E = E'$$

Correspondence

The following theorem states that CAP standard semantics and its non standard semantics are in strong bisimulation. They share equivalent initial states and each possible set of transitions any state in the non standard semantics (respectively in the standard one) is computable in the standard one (respectively in the non standard one).

In the following we denote by Π the function that maps a non standard configuration to its standard flavor. Its definition is given in Section A.1, page 215.

Theorem 3.1 *Let \mathcal{S} be a CAP term and C_0 its associated non standard initial configuration. We have $\mathcal{S} \equiv \Pi(C_0)$, and for each non standard configuration C and for each word $u \in (\mathcal{L}^2 \cup \mathcal{L}^3)^*$ such that $C_0 \xrightarrow{u}_{ns}^* C$:*

1. $\forall \lambda \in (\mathcal{L}^2 \cup \mathcal{L}^3), C \xrightarrow{\lambda}_{ns} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;
2. $\forall \lambda \in (\mathcal{L}^2 \cup \mathcal{L}^3), \Pi(C) \xrightarrow{\lambda} P \implies \exists D, \begin{cases} C \xrightarrow{\lambda}_{ns} D \\ \Pi(D) \equiv P \end{cases}$.

Proof 3.2 *The proof is presented in the appendix, Section A.1, page 215.*

3.3 ABSTRACTING NON STANDARD SEMANTICS

Once the non standard flavor of our calculus has been defined, we now rely on abstract interpretation to over-approximate the semantics of terms.

We now consider abstract domains used to over-approximate the collecting semantics of terms. Elements of such domains are abstract representations of sets of non standard configurations.

In a first part, we give the abstract domain requirements in order to soundly over-approximate the collecting semantics of terms. These definitions are later used when we define our own CAP specific abstract domains.

The next two parts give an insight on generic abstract domains proposed by FERET in [49]² These domains are not CAP specific and can be applied to any encoded calculus. When building an analysis, we construct abstract domains using standard domain operators such as the coalescent product of domains. The presented abstract domains are then a basis of our final main abstract domain construction. The first one addresses control flow properties while the second one counts occurrences of threads in configurations.

² We only present the minimal subset required for our work. We redirect the reader to FERET Ph.D. thesis for an extensive description.

3.3.1 Abstracting collecting semantics

Concrete collecting semantics

We aim to automatically check properties on mobile terms described as non standard encodings.

Let \mathcal{L}_p be a set of program points. Let Σ be a set of transition labels built over \mathcal{L}_p . We denote by \mathcal{C} the set of non standard configurations with program points in \mathcal{L}_p .

We over-approximate the collecting semantics of an initial configuration $C_0 \in \mathcal{C}$ as the least fixed point of the \cup -complete endomorphism on the complete lattice $\wp(\Sigma^* \times \mathcal{C})$:

$$\mathbb{F}(X) = (\{\epsilon\} \times C_0) \cup \left\{ (u.\lambda, C') \mid \exists C \in \mathcal{C}, (u, C) \in X \text{ and } C \xrightarrow{\lambda} C' \right\}$$

Abstract domain requirements

We introduce the definition of an abstraction for sets of non standard terms. It is presented as a concretization-based abstract interpretation as mentioned in Section 2.2.3.

Definition 3.3 *An abstraction is a tuple*

$$\mathcal{A} = (\mathcal{C}^\#, \sqsubseteq^\#, \sqcup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightsquigarrow, \nabla)$$

that satisfies the following assumption:

1. $(\mathcal{C}^\#, \sqsubseteq^\#)$ is a pre-order;
2. $\sqcup^\#$ is such that $\forall a, b \in \mathcal{C}^\#, a^\# \sqsubseteq^\# a \sqcup^\# b$ and $b^\# \sqsubseteq^\# a \sqcup^\# b$;
3. $\perp^\# \in \mathcal{C}^\#$ satisfies $\forall a \in \mathcal{C}^\#, \perp^\# \sqsubseteq^\# a$;
4. $\gamma : \mathcal{C}^\# \rightarrow \wp(\Sigma^* \times \mathcal{C})$ is a monotonic map;
5. $C_0^\# \in \mathcal{C}^\#$ is such that $\{\epsilon\} \times \mathcal{C}_0 \subseteq \gamma(C_0^\#)$;
6. $\rightsquigarrow \in \wp(\mathcal{C}^\# \times \Sigma \times \mathcal{C}^\#)$ is an abstract counterpart of the non standard transition relation. It satisfies

$$\forall C^\# \in \mathcal{C}^\#, \forall (u, C) \in \gamma(C^\#), \forall \lambda \in \Sigma, \forall C' \in \mathcal{C},$$

$$C \xrightarrow{\lambda} C' \implies \exists C'^{\#} \in \mathcal{C}^\#, (C^\# \rightsquigarrow C'^{\#}) \text{ and } (u.\lambda, C') \in \gamma(C'^{\#}).$$

7. ∇ is a widening operator. It must satisfy the property of the union operator, $\sqcup^\#$, as well as provide a guaranty to converge in a finite number of iterations: $\forall (e_n^\#)_{n \in \mathbb{N}} \in (\mathcal{C}^\#)^{\mathbb{N}}$, the sequence $(e_n^\nabla)_{n \in \mathbb{N}}$ defined as:

$$\begin{cases} e_0^\nabla &= e_0^\# \\ e_{n+1}^\nabla &= e_n^\nabla \nabla e_{n+1}^\# \end{cases}$$

is ultimately stationary.

The set $\mathcal{C}^\#$ is the abstract domain. It denotes the set of properties satisfied by a set of reachable terms. The pre-order allows us to compare elements. The greater an element is in the pre-order, the less precise information it contains. The union operator $\sqcup^\#$ allows to build new elements, merging the data of two abstract elements to denote the properties satisfied by both. The abstract basis $\perp^\#$ is the bottom element of our pre-order, it is used as the basis of our iteration computations. The concretization function γ associates a set of concrete elements to abstract elements modeling properties. It is used to guarantee the soundness of the abstraction. Then the element $C_0^\#$ and the transition relation \rightsquigarrow are abstract counterparts of the initial element and the non standard transition relation, respectively. Finally the ∇ operator is used to ensure convergence of the fixed point computation in case of abstract domains that contain infinite sequences of totally ordered elements.

Abstract collecting semantics

Given an abstraction as defined in Definition 3.3, the abstract counterpart $\mathbb{F}^\#$ of the concrete collecting semantics \mathbb{F} can then be defined:

$$\mathbb{F}^\#(C^\#) = \bigsqcup^\# (\{C'^\# \mid \exists \lambda \in \Sigma, C^\# \rightsquigarrow^\lambda C'^\#\} \sqcup \{C_0^\#\})$$

Following the above assumptions, we have

$$\forall C^\# \in \mathcal{C}^\#, \mathbb{F} \circ \gamma(C^\#) \subseteq \gamma \circ \mathbb{F}^\#(C^\#)$$

Furthermore, as \mathbb{F} is a \cup -complete morphism, the KLEENE constructive version of TARSKI fixed point theorem gives us that the least fixed point of $\mathbb{F}^\#$ can be computed as $\text{lfp}_\emptyset \mathbb{F} = \bigcup \{\mathbb{F}^n(\emptyset), n \in \mathbb{N}\}$.

As a consequence, we obtain the soundness of our analysis:

Theorem 3.4 $\text{lfp}_\emptyset \mathbb{F} \subseteq \bigcup \{\gamma \circ \mathbb{F}^{\#n}(\perp^\#), n \in \mathbb{N}\}$

The use of the ∇ operator does not break the result but rather ensure to compute $\text{lfp} \mathbb{F}^\#$ in a finite number of iterations.

3.3.2 *Approximating control flow*

A first kind of abstraction that is crucial in this framework is the definition of abstract domains to represent control flow information in threads. These abstract domains allow to soundly over-approximate transitions. They contain the (abstract) information about thread markers and their environment data. Their use under a usual product of abstract domains allows to rely on these domains to drive transitions. The chapter [49, Chap. 8] addresses such domains. We implemented them and use them as basic abstract domains in our framework. The operational semantics for control flow approximation is also generic and can be instantiated with many domains. We sketch here its semantics and then describe two of its instantiations.

Abstract operational semantics

The abstract domain for control flow approximation approximates information by program point. It associates to a given program point $p \in \mathcal{L}_p$ an abstract representation of all threads (p, id, E) that have been present in reachable states from the initial term.

This abstract domain approximates variable values of thread environments as well as their marker for a given set of configurations. It is parametrized by an abstract domain called an Atom Domain. We associate to each program point an atom which describes the values of variables and markers of the threads that can be associated to this program point.

When computing an interaction, we merge the interacting *atoms* associated to the interacting threads (primitive reagents[#]), computing a *molecule*, and add synchronization constraints (primitive sync[#]).

If they are satisfiable, *i.e.* the *molecule* is different from \perp , the interaction is possible. We then compute the value passing and the marker computation (function *marker_value*) on the *molecule*. New launched threads (primitive launch[#]) are created by building an *atom* for each of them, projecting the *molecule* information on the environment associated to each launched thread. Finally we update the *atom* of each program point by computing its union with the appropriate resulting atom.

In this generic domain, we only focus on values, so we completely abstract away occurrences of threads and thus deletion of interacting threads.

The main *atom* domain we use is a product of three domains. The first two represent equalities and disequalities among values and markers using graphs, the third one approximates the shape of markers and values with an automaton.

The operational semantics is given in Figure 3.4. It relies on several primitives defined in the *atom* abstract domain.

Theorem 3.5 *The control flow abstraction satisfies the soundness assumption of Definition 3.3.*

Proof 3.6 *The proof can be found [49, App. D].*

Equality and disequality relations among variables and markers

A first kind of abstract domain is the domain of equalities and disequalities. In this domain, elements are graphs built over non empty sets of variable names of program point thread environment. The set of variable names is partitioned among nodes. A node denotes variables sharing the same value. An edge between two nodes denotes a disequality relation between variables associated to the different nodes.

The \perp element is the graph containing a single node with all variable names in it and an edge from the node to itself. The abstract union may split partitions

Figure 3.4 Abstract operational semantics for control flow approximation.

Let $C^\# \in \mathcal{C}^{env}$ be an abstract configuration. Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_passing)$ be a formal rule. Let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program point labels and $(pi^k)_{1 \leq k \leq n} = (s^k, (par^k), (bd^k), cont^k)$ be a tuple of partial interactions. We denote by $I(p^k)$ the interface of program point p^k , *i.e.* the set of variable names its environment must define. We define mol by $\text{reagents}^\#((p^k)_k, (par^k)_k, \text{compatibility}, C^\#)$.

When

1. $\forall k \in \llbracket 1; n \rrbracket, pi_k \in \text{interaction}(p^k)$;
2. and $mol \neq \perp_{(I(p^k))_k}$.

Then

$$C \xrightarrow{\mathcal{R}, (p^k)_k, (pi^k)_k}_{env} \bigsqcup \{C; \text{new_threads}\}$$

Where

1. $mol' = \text{marker_value}((p^k)_k, mol, (bd^k)_k, (par^k)_k, v_passing)$;
2. $\text{new_threads} = \text{launch}^\#((p^k, cont^k)_k, mol')$.

into smaller ones and remove edges. The less precise element being the graph with one singleton node for each variable and no edges.

When computing a transition, the nodes are labeled by an index in order to differentiate their origin. The graphs associated to the different interacting thread program points are gathered in a single *molecule* graph. Synchronization constraints are represented in it, merging nodes sharing equal variable values. If the resulting molecule is valid, *i.e.* no edge from a node to itself, the transition can occur.

The value passing introduces new variables in the graph. Their value is also modeled with equality constraints, adding new variable names in nodes that carry their value. Finally each thread is launched by projecting the *molecule* graph onto the variables of the thread environment, removing added index.

This domain can be instantiated to abstract relations either between variable values in thread environments or marker values in threads. In the latter case, an additional I variable is considered that represents the thread marker.

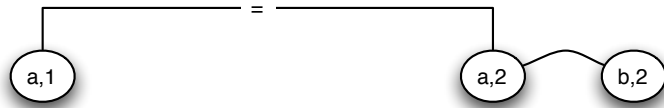
The Figure 3.5 illustrates the domain use on the first transition of the Example 2.6 presented above in its non standard encoding.

Figure 3.5 Transition computation for the abstract domain of equalities and disequalities among variables.

The initial abstract element is the following, associating a graph element to program points 1 and 6.



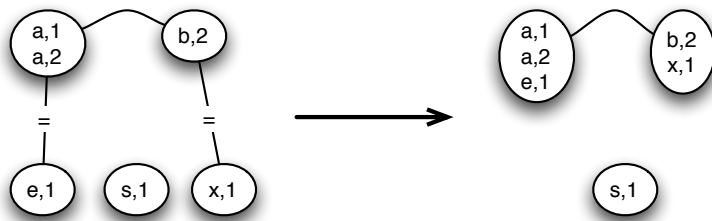
The atom gathering gives:



The synchronization introduces equalities between nodes.



Then the value passing is computed for new variables: $\{(e, 1), (s, 1), (x, 1)\}$.



Finally the molecule is projected onto each launched thread interface. In our case, a thread at program point 5:



Regular approximation

The preceding domains allow to carry relations between variables but they are not able to give any information about the actual values of variables, *i.e.* at least their binder. The domain of regular approximation solves this problem. It is a simple automaton over program points denoting words over \mathcal{L}_p .

We recall that environment values are pairs $(p, id) \in \mathcal{L}_p \times \mathcal{M}$ where $\mathcal{M} = \mathcal{L}_p^*$ denotes history marker letter. The regular approximation over-approximates the possible values by abstracting words $p.id \in \mathcal{L}_p^*$.

Initial letters denote possible binders and the words from these initial letters to final ones are abstractions of possible history markers.

As in the above domain on equalities and disequalities among markers, we introduce a special variable I that denotes the thread marker. In that case, the word over-approximates directly the value id , *i.e.* without any binder as a prefix.

We now give the formal definitions of this abstraction since we use its information in our later analyses. We do not detail the transitions but rather the main domain primitives and the concretization.

ABSTRACT DOMAIN OF REGULAR APPROXIMATION We introduce the domain Reg_{Σ_m} of regular approximation of words on Σ_m . An automaton on the alphabet Σ_m is defined by a tuple (i, f, t, b) where $i, f \in \wp(\Sigma_m)$ denote respectively the set of first and last letter of words recognized by the automaton. $t \in (\Sigma_m \rightarrow \wp(\Sigma_m))$ is a map that associates to each letter of Σ_m its successors. Finally the boolean $b \in \{0, 1\}$ describes whether the automaton recognizes the empty word.

The concretization function $\gamma_{\Sigma_m}^{\text{Reg}}$ associates to each automaton $(i, f, t, b) \in \text{Reg}_{\Sigma_m}$, the set of words u built on Σ_m such that:

$$\left\{ \begin{array}{l} |u| > 0 \Rightarrow u_1 \in i \\ |u| > 0 \Rightarrow u_{|u|} \in f \\ \forall i \in \llbracket 1; |u| \rrbracket, u_{i+1} \in t(u_i) \\ |u| = 0 \Rightarrow b = 1 \end{array} \right.$$

As is, two elements Reg_{Σ_m} can recognize the same language while having different sets i, f and different maps t . In fact, a letter could be present in i , not in f and without any successor by t . This letter is useless in the representation. The reduction operator ρ^{Reg} allows to obtain a canonical and minimal representation of the automaton. It combines a forward reachability computation with a backward one in order to reduce i, f and t .

The domain Reg_{Σ_m} is fitted with a complete lattice structure $(\text{Reg}_{\Sigma_m}, \sqsubseteq_{\Sigma_m}^{\text{Reg}}, \sqcap_{\Sigma_m}^{\text{Reg}}, \sqcup_{\Sigma_m}^{\text{Reg}}, \perp_{\Sigma_m}^{\text{Reg}}, \top_{\Sigma_m}^{\text{Reg}})$

- $\sqsubseteq_{\Sigma_m}^{\text{Reg}}, \sqcap_{\Sigma_m}^{\text{Reg}}$ and $\sqcup_{\Sigma_m}^{\text{Reg}}$ are defined component-wise from usual set operations. The intersection is reduced with the ρ^{Ref} operator.
- $\perp_{\Sigma_m}^{\text{Reg}} = (\emptyset, \emptyset, [\lambda \mapsto \emptyset], 0)$;
- $\top_{\Sigma_m}^{\text{Reg}} = (\Sigma_m, \Sigma_m, [\lambda \mapsto \Sigma_m], 1)$.

In order to manipulate elements of the domain, the primitive $\text{PREFIX}_{\Sigma_m}^{\text{Reg}}$ is defined, adding a letter as a prefix of the language recognized by the automaton.

ATOM The domain Shape that relies on $\text{Reg}_{\mathcal{L}}$ is introduced to abstract values and markers. Each interface V is associated to elements of the domain $\text{Atom}_V^{\text{Shape}} = (V \cup \{I\} \rightarrow \text{Reg}_{\mathcal{L}})$. The automaton associated to the variable I abstracts the shape of thread markers while the automaton associated to interface variables recognize languages $\{l.m\}$ where (l, m) is a value of a variable.

More formally, each element env of the domain is mapped to the set:

$$\gamma_V^{\text{Shape}}(\text{env}) = \gamma_{\mathcal{L}}^{\text{Reg}}(\text{env}(I)) \times \prod_{v \in V} \{(l, \text{id}) \mid l.\text{id} \in \gamma_{\mathcal{L}}^{\text{Reg}}(f(v))\}$$

Operators $\sqsubseteq_V^{\text{Shape}}, \sqcup_V^{\text{Shape}}$ are defined component-wise using $\sqsubseteq_{\mathcal{L}}^{\text{Reg}}$ and $\sqcup_{\mathcal{L}}^{\text{Reg}}$. Similarly, \perp_V^{Shape} is defined by $\perp_{\mathcal{L}}^{\text{Reg}}$.

The Figure 3.6 illustrates the domain use on the first transition of the Example 2.6 presented above in its non standard encoding.

3.3.3 Occurrence counting

The previous part addressed the over-approximation of control flow property but did not address any property related to occurrences of threads in configuration. The interacting thread consumption was not even modeled. We can now rely on the control flow information to drive feasible transitions and to compute value passing. Now we can focus on specific occurrence counting properties.

We now over-approximate the occurrences of threads in configurations as well as the possible occurrences of transition labels in the words of transitions since the initial configuration.

In this generic domain, we count both threads associated to a particular program point and transition labels, the set of which is denoted by \mathcal{V}_c . As in the marker approximation, we abstract transition labels by only one program point. In our case, we take the program point denoting the behavior of the actor, either a behavior thread program point or a statically defined actor program point. In order to differentiate program points denoting threads from program points denoting transitions, we tag them with a boolean: $\mathcal{V}_c = \mathcal{L}_p \times \mathbb{B}$.

We first approximate the non standard semantics by the domain $\mathbb{N}^{\mathcal{V}_c}$, associating to each program point its thread occurrence in the configuration and

Figure 3.6 Transition computation for the abstract domain of regular approximation of shape.

The initial abstract element is

$$1 \left\{ \begin{array}{l} I \quad (\emptyset, \emptyset, \emptyset, T) \\ a \quad (\alpha, \alpha, \emptyset, F) \end{array} \right. \quad 6 \left\{ \begin{array}{l} I \quad (\emptyset, \emptyset, \emptyset, T) \\ a \quad (\alpha, \alpha, \emptyset, F) \\ b \quad (\beta, \beta, \emptyset, F) \end{array} \right.$$

The atom gathering gives

$$\begin{array}{ll} I, 1 & (\emptyset, \emptyset, \emptyset, T) & a, 1 & (\alpha, \alpha, \emptyset, F) \\ I, 2 & (\emptyset, \emptyset, \emptyset, T) & a, 2 & (\alpha, \alpha, \emptyset, F) \\ b, 2 & (\beta, \beta, \emptyset, F) & & \end{array}$$

The synchronization ensures that

$$a, 1 \sqcap_{\Sigma_m}^{\text{Reg}} a, 2 = (\alpha, \alpha, \emptyset, F) \neq \perp_{\Sigma_m}^{\text{Reg}}$$

Then the value passing is computed for new variables: $\{(e, 1), (s, 1), (x, 1)\}$. The marker is left untouched: the transition does not involve a replication partial interaction.

$$\begin{array}{l} e, 1 = a, 1 \sqcap_{\Sigma_m}^{\text{Reg}} a, 2 = (\alpha, \alpha, \emptyset, F) \\ s, 1 = \text{PREFIX}_{\Sigma_m}^{\text{Reg}}(1, (I, 1)) = (1, 1, \emptyset, F) \\ x, 1 = b, 2 = (\beta, \beta, \emptyset, F) \end{array}$$

Finally the molecule is projected on each launched thread interface. In our case:

$$5 \left\{ \begin{array}{l} I \quad (\emptyset, \emptyset, \emptyset, T) \\ x \quad (\beta, \beta, \emptyset, F) \\ s \quad (1, 1, \emptyset, F) \end{array} \right.$$

to each transition label its occurrence in the trace that leads to the configuration. At the level of the collecting semantics, we obtain an element in $\wp(\mathbb{N}^{\mathcal{V}_c})$. We then abstract such a domain by a domain $\mathcal{N}_{\mathcal{V}_c}$ which is a reduced product between the domain of intervals indexed by \mathcal{V}_c and the domain of affine equalities [68] built over \mathcal{V}_c .

When computing a transition, we check that the occurrences of interacting threads are sufficient to allow it (primitive $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}$). If we do not obtain the bottom element of our abstract domain, *i.e.* the synchronization constraint is satisfiable, we add (primitive $+^\#$) the new transition label, the launched threads (primitives $\beta^\#$ and $\Sigma^\#$) and remove (primitive $-^\#$) consumed threads.

This domain formal definition can be found in [47] in a π calculus context and in [49] in a more generic view.

Abstract domain $\mathbb{N}^{\mathcal{V}_c}$

A first approximation of a reachable configuration is given using the abstraction $\Pi_{\mathbb{N}^{\mathcal{V}_c}}$ which maps each pair $(\mathbf{u}, C) \in \Sigma^* \times \mathcal{C}$, to the family of natural numbers indexed by \mathcal{V}_c , defined as:

$$(\Pi_{\mathbb{N}^{\mathcal{V}_c}}(\mathbf{u}, C))_{\mathbf{v}} = \begin{cases} \text{Card}(\{(p, \text{id}, E) \in C \mid p = \mathbf{v}\}) & \text{when } (\mathbf{v}, T) \in \mathcal{V}_c \\ |\psi(\mathbf{u})|_{\mathbf{v}} & \text{when } (\mathbf{v}, F) \in \mathcal{V}_c \end{cases}$$

where $\psi(\mathbf{u})$ denotes the projection of the transition labels onto \mathcal{L}_p^* .

Each thread program point is then associated to its occurrence number in the configuration C and each transition label to its occurrence number in \mathbf{u} .

The concretization of a set $A^\#$ of natural numbers indexed by a set \mathcal{V}_c is then defined:

$$\gamma_{\mathbb{N}^{\mathcal{V}_c}} = \left\{ (\mathbf{u}, C) \in \Sigma^* \times \mathcal{C} \mid \Pi_{\mathbb{N}^{\mathcal{V}_c}}(\mathbf{u}, C) \in A^\# \right\}$$

Abstract domain $\mathcal{N}_{\mathcal{V}_c}$

The set of natural numbers indexed by \mathcal{V}_c is now abstracted using the domain $\mathcal{N}_{\mathcal{V}_c}$. This domain is generic and must defined the primitives used to specify its semantics. These primitives must satisfy the following constraints in order to keep the domain sound:

Definition 3.7 (Soundness requirements for semantic primitives of $\mathcal{N}_{\mathcal{V}_c}$)

1. *it must be fitted with a pre-order relation;*
2. *the bottom element $\perp_{\mathcal{N}_{\mathcal{V}_c}}$ must verify $\gamma_{\mathcal{N}_{\mathcal{V}_c}}(\perp_{\mathcal{N}_{\mathcal{V}_c}}) = \emptyset$ and $\forall \mathbf{a} \in \mathcal{N}_{\mathcal{V}_c}, \perp_{\mathcal{N}_{\mathcal{V}_c}} \sqsubseteq_{\mathcal{N}_{\mathcal{V}_c}}^\# \mathbf{a}$. The concretization is then strict;*

3. *the union must be sound:* $\forall A \in \wp_{\text{finite}}(\mathcal{N}_{\mathcal{V}_c}), \cup^{\#}_{\mathcal{N}_{\mathcal{V}_c}}(A) \in \mathcal{N}_{\mathcal{V}_c}$ and $\forall a \in A, a \sqsubseteq^{\#}_{\mathcal{N}_{\mathcal{V}_c}} \cup^{\#}_{\mathcal{N}_{\mathcal{V}_c}}(A)$;
4. *a widening operator $\nabla_{\mathcal{N}_{\mathcal{V}_c}}$ must be defined when needed to ensure convergence in a finite number of iteration. It must satisfy the assumption of Definition 3.3;*
5. *the addition $+^{\#}$ of two families of natural numbers satisfies $\forall a^{\#}, b^{\#} \in \mathcal{N}_{\mathcal{V}_c}, a^{\#} +^{\#} b^{\#} \in \mathcal{N}_{\mathcal{V}_c}$ and $\{(a_v + b_v)_{v \in \mathcal{V}_c} \mid a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\#}), b \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(b^{\#})\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\#} +^{\#} b^{\#})$. It is the abstract counterpart of the natural number binary addition;*
6. *abstract subtraction $-^{\#}$ is similarly defined:*
 $\forall a^{\#}, b^{\#} \in \mathcal{N}_{\mathcal{V}_c}, (a^{\#} -^{\#} b^{\#}) \in \mathcal{N}_{\mathcal{V}_c}$ and

$$\left\{ (a_v - b_v)_{v \in \mathcal{V}_c} \mid \begin{array}{l} a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\#}), b \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(b^{\#}), \\ \forall v \in \mathcal{V}_c, a_v \geq b_v \end{array} \right\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\#} -^{\#} b^{\#})$$
7. *the synchronization condition enforces the presence of interacting threads:* $\forall a^{\#} \in \mathcal{N}_{\mathcal{V}_c}, t \in \mathbb{N}^{\mathcal{N}_{\mathcal{V}_c}}, \text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, a^{\#}) \in \mathcal{N}_{\mathcal{V}_c}$ and
 $\{a \mid a \in \gamma_{\mathcal{N}_{\mathcal{V}_c}}(a^{\#}), a_v \geq t_v, \forall v \in \mathcal{V}_c\} \subseteq \gamma_{\mathcal{N}_{\mathcal{V}_c}}(\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, a^{\#}))$;
8. *the primitives $0_{\mathcal{N}_{\mathcal{V}_c}}$ and $1_{\mathcal{N}_{\mathcal{V}_c}(v)}$ represent families of natural numbers indexed by \mathcal{V}_c that are respectively all equal to 0 in $0_{\mathcal{N}_{\mathcal{V}_c}}$, and all equal to 0 except v equal to 1 in $1_{\mathcal{N}_{\mathcal{V}_c}(v)}$.*

Three other primitives are automatically defined using these basic primitives. $\Sigma^{\#}$ computes the abstract summation of abstract elements using the abstract binary addition; $\chi^{\#}(V)$ associates 1 to each element of V and 0 to other variables; finally $\beta^{\#}$ is used to build abstract continuations; it maps continuations, sets of static threads obtained using the extraction function, to abstract elements associating occurrence 1 to variables $(p, 0) \in \mathcal{V}_c$ associated to launched threads.

Operational semantics

The operational semantics relies on the above primitives.

The initial state is obtained by calling $\beta^{\#}$ on the set of initial static threads init_s .

$$C_0^{\mathcal{N}_{\mathcal{V}_c}} = \beta^{\#}(\text{init}_s)$$

An abstract transition is computed in four steps:

- find interacting threads;
- ensure their presence in the current abstract element;

Figure 3.7 Abstract operational semantics for occurrence counting abstraction.

Let $C^\#$ be an abstract configuration, let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program points label and $(pi^k)_{1 \leq k \leq n} = (s^k, (par^k), (bd^k), cont^k)$ be a tuple of partial interactions. We define the tuple $t \in \mathbb{N}^{\mathcal{V}_c}$ so that t_v be the occurrence of v in $(p^k)_{1 \leq k \leq n}$.

When

1. $\forall k \in \llbracket 1; n \rrbracket, pi^k \in \text{interaction}(p^k)$;
2. and $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, C^\#) \neq \perp_{\mathcal{N}_{\mathcal{V}_c}}$.

Then

$$C \xrightarrow{\mathcal{R}, (p^k)_k, (pi^k)_k} \# \text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, C^\#) + \# \text{Transition} + \# \text{Launched} - \# \text{Consumed}$$

Where

1. $\text{Transition} = 1_{\mathcal{N}_{\mathcal{V}_c}}(\psi((p^k)_k))$ where ψ projects $\mathcal{L}_p^2 \cup \mathcal{L}_p^3$ to \mathcal{L}_p ;
 2. $\text{Launched} = \Sigma^\#((\beta^\#(cont^k))_k)$;
 3. $\text{Consumed} = \Sigma^\#(1_{\mathcal{N}_{\mathcal{V}_c}}(p^k))_{k \in \llbracket 1; n \rrbracket \wedge \text{type}(s_{k'}) \neq \text{replication}}$.
-

- compute possible continuations;
- launch new threads, remove interacting ones of type computation, and add the program point extracted from the transition label.

The Figure 3.7 gives a more detailed definition of the abstract transition computation.

Theorem 3.8 *The abstraction defined by $(\mathcal{N}_{\mathcal{V}_c}, \sqsubseteq^\#_{\mathcal{N}_{\mathcal{V}_c}}, \cup^\#_{\mathcal{N}_{\mathcal{V}_c}}, \perp^\#_{\mathcal{N}_{\mathcal{V}_c}}, \gamma_{\mathbb{N}^{\mathcal{V}_c}} \circ \gamma_{\mathcal{N}_{\mathcal{V}_c}}, C_0^{\mathcal{N}_{\mathcal{V}_c}}, \rightsquigarrow_{\mathcal{N}_{\mathcal{V}_c}}, \nabla_{\mathcal{N}_{\mathcal{V}_c}})$ is a sound abstraction with respect to Definition 3.3.*

Proof 3.9 *The proof can be found [49, App. D].*

Non relational abstraction

A first abstract domain to represent occurrences is the interval abstract domain. It associates to each thread program point and to each transition label its over-approximation as an interval of positive integers.

The synchronization primitive ensures that at least one of each interacting thread is present. More precisely, that the over-approximation of yet computed transitions has led to the creation of at least one of each interacting thread.

Abstract transition computation increases by one the interval associated to the program point of every launched thread and decreases by one the interval associated to the program point of each non replicating interacting thread. When computing the union of the abstract elements before and after the abstract transition computation, we obtain the over-approximation of the occurrence number of threads in reachable configurations.

The abstract transition also increases the transition label by one.

Relational abstraction

A second domain, a relational one, carries information of relational properties between occurrences of threads and transition labels. We use the domain of affine equalities as defined by KARR in [68].

Elements of this domain are mapped to elements of $\wp(\mathbb{N}^{\mathcal{V}_c})$ by the monotonic morphism $\gamma_{\mathcal{K}}$ that associates to each affine system its set of solutions.

The union $a \cup_{\mathcal{K}} b$ computes the least affine space containing the two affine spaces a and b . This lattice does not allow infinite ascending chains, it is bounded by the number of dimensions, *i.e.* the size of \mathcal{V}_c . Therefore the ∇ operator is not necessary and is mapped to the union operator.

The abstract addition and abstract subtraction are the natural addition and subtraction of matrices.

Finally a specific point of this domain is the absence of constraints for the synchronization phase of the abstract semantics. It relies on the one computed by the non relational abstraction.

Approximated reduction

Finally the main occurrence counting abstract domain proposed is a reduction between the relational abstract domain and the non relational abstract domain. It is a reduced product of these two domains. We do not detail precisely the reduction operator as its enhancement is the topic of a next chapter (cf. Chapter 5).

The idea is to constrain intervals associated to both transition labels and thread program points by the relational information we have between them. The reduction operator algorithm is in two steps. The first one tries to bound infinite intervals. The second one tries to narrow interval bounds using relational information and interval propagation.

3.4 DISCUSSION

In this chapter, we introduced the generic framework of FERET [49] and its instantiation for CAP. Analyzing the collecting semantics of a CAP term is now done in two steps.

ENCODING CAP IN NON STANDARD SEMANTICS We first automatically annotate the term with program points and construct the different sets and functions describing its encoding in the non standard semantics.

The non standard representation of the term is based on an explicit representation of transitions history that led to the creation of the different values and processes. In the CAP instantiation of the framework, values can either denote addresses created by a ν binder or denote behaviors associated to an existing actor in the past of the configuration. This behavior can also be a recursive instance of this past actor. We choose to model behaviors as definition threads in order to allow the modeling of the high-order feature of CAP.

The presented encoding deals with the full version of CAP without any restriction on the shape of terms. In particular, it handles properties specific to CAP which were not tackled in FERET encodings of other calculi: asynchronous communications, high-orderness, and non uniform interfaces.

OVER-APPROXIMATING THE COLLECTING SEMANTICS The second step of the analysis is now the over-approximation of the collecting semantics of the encoded term relying on abstract interpretation by means of abstract domains. The generic abstractions proposed by FERET and presented above can already be applied on CAP encoding.

Control flow properties For example, the control flow abstract domain can give precise information on the binders associated to the different address variables. We can also identify which are the variables that denote addresses and which ones denote behaviors. In the later case, we have an over-approximation of the associated behaviors as well as their internal variable assignments.

An interesting feature of this kind of approach is the precision in the use of continuations. In other approaches like type systems, we cannot differentiate the different behavior branches with a small complexity cost. Removing dead branches from the result requires an incremental typing or the use of dependent types which may be costly and complex.

In this approach, each transition matches a particular branch and the analysis gives precise results on the subset of the behaviors used.

Occurrence counting properties The result of the occurrence counting abstract domain is also very interesting. Usual type system-based approaches have a lot of difficulties to count. The use of numerical abstract domains as presented here is of great help for such matter. Using the result of occurrence counting abstract domain, we can bound the number of messages, of actors, of transitions computed using a given actor, even the number of defined behaviors.

The relational part can also give us mutual exclusion properties between threads or between threads and transitions. For example, one can obtain that a

thread is present until a transition of a particular label is computed. In case of non-terminating systems, we can obtain that the number of computation transitions for a given label is equal to the number of messages produced. In that case, the system grows infinitely but any produced message can be handled under fairness assumption.

OBSERVABLE PROPERTIES We now present some properties that one can observe in the analysis of terms.

Bounded resources In the following example, our analysis is able to find that at most one message is present in the system: program points 3, 7 and 9 are associated to interval $\llbracket 0; 1 \rrbracket$. The system described by this term is bounded. Furthermore, we have the constraint $p_1 + p_4 + p_8 = 1$.

$$\begin{aligned} & \nu a^\alpha, \nu b^\beta, \quad a \triangleleft^{1:\llbracket 0; 1 \rrbracket} \text{ping}() \\ & \quad \parallel a \triangleright^{2:\llbracket 0; 1 \rrbracket} [\text{ping}^{3:\llbracket 1; 1 \rrbracket}() = \zeta(e, s)(b \triangleleft^{4:\llbracket 0; 1 \rrbracket} \text{pong}() \parallel e \triangleright^{5:\llbracket 0; 1 \rrbracket} s)] \\ & \quad \parallel b \triangleright^{6:\llbracket 0; 1 \rrbracket} [\text{pong}^{7:\llbracket 1; 1 \rrbracket}() = \zeta(e, s)(a \triangleleft^{8:\llbracket 0; 1 \rrbracket} \text{ping}() \parallel e \triangleright^{9:\llbracket 0; 1 \rrbracket} s)] \end{aligned}$$

In addition, we can also detect whether a system does not generate an unbounded number of actors present at the same time in a given configuration.

$$\begin{aligned} & \nu a^\alpha, \quad a \triangleright^{1:\llbracket 0; 1 \rrbracket} [m^{2:\llbracket 1; 1 \rrbracket}() = \zeta(e, s)(\nu b^\beta b \triangleright^{3:\llbracket 0; 1 \rrbracket} s \parallel b \triangleleft^{4:\llbracket 0; 1 \rrbracket} m())] \\ & \quad \parallel a \triangleleft^{5:\llbracket 0; 1 \rrbracket} m() \end{aligned}$$

In the preceding example, we automatically detect that the number of threads associated to program point 3 lies in $\llbracket 0; 1 \rrbracket$.

Unreachable Behaviors We are interested in determining the subset of behaviors that are never used for each set of behaviors. Due to its higher-order capability, CAP allows to send the set of behaviors syntactically associated to an actor to other actors. Therefore the use of the behavior set highly depends on the exchanged messages.

In the following example, all the branches of the behavior syntactically defined at program point 1 are used in the over-approximation. We obtain such a property by checking that each label of transition is present at least once or its continuation has been launched. *i.e.* $\forall t \in \mathcal{V}_c, \text{Inter}(t) \neq \llbracket 0; 0 \rrbracket$ where Inter is the function that maps each element of \mathcal{V}_c to its image in interval part of the analysis post fixed point.

$$\begin{aligned}
\forall a^\alpha, b^\beta, c^\gamma, \quad & a \triangleright^1 [m_0^2() = \zeta(e, s)(b \triangleleft^3 n_1(s) \parallel b \triangleleft^4 m_1(c)), \\
& m_1^5(\text{dest}) = \zeta(e, s)(\text{dest} \triangleleft^6 m_2()), \\
& m_2^7() = \zeta(e, s)(\emptyset)] \\
& \parallel b \triangleright^8 [n_1^9(\text{self}) = \zeta(e, s)(e \triangleright^{10} \text{self} \parallel c \triangleleft^{11} n_2(\text{self}))] \\
& \parallel c \triangleright^{12} [n_2^{13}(\text{self}) = \zeta(e, s)(e \triangleright^{14} \text{self})] \\
& \parallel a \triangleleft^{15} m_0()
\end{aligned}$$

We can use such an analysis to clean the term with garbage collecting-like mechanisms.

The presented abstract domains are highly generic and could be applied to any encoded calculus. However, in order to check more high level properties, we need to define abstractions that rely on CAP specifics to compute properties.

One of the necessary kind of properties we want to compute is the properties that are valid for a given address and not system wide. The next chapter, Chapter 4, proposes such an analysis by defining a partitioned abstract domain.

Another drawback of the presented abstractions is the precision of the occurrence counting one. It gives very good results but some spurious transitions are computed that should be avoided. Chapter 5 addresses such enhancements that are necessary to obtain precise properties.

The latter chapter introduces the encoding of CAP into its non standard form and proposes two analyses that can be directly applied on this encoding.

The first analysis catches dynamic properties about environment values and thread markers. It allows to distinguish recursive instances of both threads and values and to forbid some unfeasible transitions. This domain, which is also parametric and can be instantiated with different underlying domains, is essential to the global analysis of a term, as it literally drives the abstract transition computations, over-approximating, with precision, threads that could or could not interact. A drawback of it is that this domain is thread specific and does not give any information about concurrency properties, *i.e.* existence of threads that interact or are mutually exclusive, etc.

The second analysis palliates this weakness, relying on a numerical abstraction to describe occurrences of threads in configurations as well as occurrences of transition labels in traces of reachable configurations. It takes advantage of the wide set of numerical abstract domains that exist in the literature. But again, a drawback is that any occurrence counting property inferred by the analysis is global to the whole term.

Global properties are of interest when we consider the boundedness of a system, or when considering control flow properties, for example the binder that could be associated to the address of a given actor. However more precise properties are often expressed locally. An analysis such as the two preceding ones may compute locally precise data, but the collecting semantics expression merges the obtained property with the previously computed one and weakens the result. Local properties also arise when we are interested to obtain information specific to a sub-term.

In CAP, a notion that already emerges when describing the calculus, is the centering of any interaction around the targeted address. In fact, any communication involves a single address. Furthermore the main properties that we are interested in are the linearity property or the orphan freeness property, as briefly presented in Section 2.4. Both of these are inherently address centric. The first one considers the usage of addresses to install actors. The second one aims at ensuring that the sequence of behaviors associated to a given address is always able to handle sent messages. In both cases, the property has to be checked per address.

In this chapter, we propose another abstract domain that aims at partitioning computed properties by address. The first section, Section 4.1, illustrates the partitioning on CAP terms both in the concrete and in the abstract. Then

we propose in Section 4.2 a parametric partitioned abstract domain. Section 4.3 gives examples of this domain use, relying on the previously defined abstract domains. In order to support more domains, and in particular to prepare the work for the analyses presented in the next chapters, we introduce in Section 4.4 enhancements that modify the abstraction, enhancing accuracy. Finally the last section, Section 4.5 compares this abstraction to another partitioning framework proposed by FERET in [49, Chap. 10] and concludes.

4.1 PARTITIONING PROPERTIES BY ADDRESS

We now propose to partition threads given their address and later, in the abstract, given their address binder. We first propose a partitioned flavor of the CAP non standard semantics. Then we introduce the key ideas behind its abstraction.

4.1.1 Concrete address partitioning

Going back to the early definition of actors, messages sent to the same address constitute the mailbox of the actor associated to this address. A communication is then an actor that handles a message from its own mailbox.

We now redefine parts of the non standard semantics in order to make these mailboxes explicit. The primitives that are non presented in the current chapter are left untouched.

Address partitioning

A configuration is now a set of partition units indexed by a value in $\mathcal{L}_v \times \mathcal{M}$. We recall that \mathcal{L}_v denotes the set of program points of address binders and \mathcal{M} denotes the set of identity markers built over transition labels or their abstractions.

We consider a special partition unit which contains the behavior threads. In fact, these threads are our behavior definitions and are not specific to a given address.

Each partition unit is a set of threads in $\mathcal{C} = \mathcal{C}_{\text{unit}}$:

$$\mathcal{C}_{\text{unit}} = \wp(\mathcal{L}_p \times \mathcal{M} \times (\mathcal{V} \rightarrow \mathcal{L} \times \mathcal{M})).$$

We introduce the set $\mathcal{C}_{\text{part}}$:

$$\mathcal{C}_{\text{part}} = \mathcal{C}_{\text{unit}} \times ((\mathcal{L}_v \times \mathcal{M}) \rightarrow \mathcal{C}_{\text{unit}}).$$

In the following, we denote by $C_{|x}$ the subset of C threads associated to the address $x \in \mathcal{L}_v \times \mathcal{M}$. Similarly $C_{|\text{beh}}$ denotes the subset of threads of C associated to behavior program points \mathcal{L}_b .

$$(C_{|beh}, \lambda x. C_{|x}) \in \mathcal{C}_{part}$$

We now detail the different parts of the non standard semantics that are modified in this partitioned version. Used primitives that are not defined here can be found in the previous chapter.

EXTRACTING ADDRESS In order to easily obtain the address value associated to a given thread, we introduce the primitive address : $\mathcal{L}_p \times \mathcal{M} \times (\mathcal{V} \rightarrow \mathcal{L} \times \mathcal{M}) \rightarrow \mathcal{L} \times \mathcal{M}$. It is only defined on actor and message threads.

Definition 4.1 (address extraction) Let $t = (p, id, E)$ be a thread. Let $\pi_i = (s, (parameter_1), (bd_1), continuation)$ be a partial interaction. We define the primitive address as follows:

$$\begin{aligned} \text{address}(t) &\triangleq E[\text{parameter}_1] \\ \text{when } s \in &\begin{cases} \text{static_actor}_n, \\ \text{dynamic_actor}, \text{ iff } \text{exhibits}(t, \pi_i). \\ \text{message}_n \end{cases} \end{aligned}$$

SYNCHRONIZATION The synchronization primitive `sync` is modified to explicitly restrict interacting actor and message to be part of the same partition unit.

Definition 4.2 (synchronization) Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \leq k \leq n} = (p^k, id^k, E^k)$ be a n -tuple of interacting threads. We denote by p^a and p^m the actor thread and message thread in (t^k) , respectively. Let $(parameter_1)^k$ be a n -tuple of parameters and `compatibility` be a set of synchronization constraints. We denote by `sync` the original synchronization primitive as defined in the non standard semantics. The relation `syncpart` is defined as follows:

$$\begin{aligned} \text{sync}_{part}((t^k), ((parameter_1)^k), \text{compatibility}) &\triangleq \\ &\text{address}(t^a) = \text{address}(t^m) \wedge \\ &\text{sync}((t^k), ((parameter_1)^k), \text{compatibility}) \end{aligned}$$

The introduced constraint is redundant since each of our two formal rules contains a synchronization constraint that expresses that both actor and message must be bound to the same address. However, we make it more explicit using the address primitive.

REMOVING INTERACTING THREADS Removed threads are interacting threads that are not behavior definitions. Again they are all part of the same interacting partition unit. We build the partitioned set of threads to be removed, using the primitive $\text{remove}_{\text{part}}$.

Definition 4.3 (removing interacting threads) Let $n \in \mathbb{N}$ be an integer. Let $(t^k)_{1 \leq k \leq n}$ be a n -tuple of threads. Let $(\text{type}^k)_{1 \leq k \leq n}$ be a n -tuple of partial interaction types. We denote by t^a the actor thread in $(t^k)_k$, i.e. $\text{type}^a \in \{\text{static_actor}_n, \text{dynamic_actor}_n\}$. The primitive $\text{remove}_{\text{part}}$ is defined as follows:

$$\text{remove}_{\text{part}}\left((t^k)_{1 \leq k \leq n}, (\text{type}^k)_{1 \leq k \leq n}\right) \triangleq \\ \text{address}(t^a) \mapsto \{t^k \mid 1 \leq k \leq n \wedge \text{type}^k \neq \text{replication}\}$$

LAUNCHING NEW THREADS Launching new threads follows the same line: the set of static threads associated to each continuation is updated using value passing and marker computation. Then the obtained set of threads is split among partition units depending on the associated address. New behavior threads are put apart outside partition units to be easily added to the set of behavior definitions $C_{|\text{beh}}$.

It relies on the primitive update that associates the new computed marker to variables defined in each continuation static thread environment.

Definition 4.4 (continuation launching) Let $\text{Ct} \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ be a set of continuations. Let $\text{id} \in \mathcal{M}$ be a history marker and let $E \in \wp(\mathcal{V} \rightarrow \mathcal{L} \times \mathcal{M})$ be an environment. We define the $\text{launch}^{\text{part}}$ primitive as:

$$\text{launch}^{\text{part}}(\text{Ct}, \text{id}, E) \triangleq (\text{beh}, \text{part})$$

where

- $\text{launched} = \left\{ (p, \text{id}, \bar{E}) \left| \begin{array}{l} I(p) \text{ the interface of program point } p, \\ (p, E_s) \in \text{Ct} \end{array} \right. \right\};$
- $\bar{E} = (\text{update}(\text{id}, E, E_s))_{|I(p)}$;
- $\text{beh} = \{(p, \text{id}, E) \in \text{launched} \mid p \in \mathcal{L}_b\}$;
- $\text{part} = \lambda x. \{t \in \text{launched} \mid x = \text{address}(t)\}$.

In the launched set of threads, some threads may be associated to an address which has just been created by the update primitive. In that case, a new partition unit is created.

Operational semantics

The operational semantics is detailed in Figure 4.1. The initial configuration is obtained as before, by launching the initial static thread with an empty marker. This launching associates to each initial thread its adequate partition unit. Initial behavior threads have to be kept separate.

Figure 4.1 CAP non standard partitioned semantics.

Let (c_b, part) be a partitioned configuration. Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, \text{v_passing})$ be a formal rule. Let us be given a tuple $(t^k)_{1 \leq k \leq n} = (p^k, \text{id}^k, E^k)_{1 \leq k \leq n} \in C^n$ of distinct threads and a tuple $(\text{pi}^k)_{1 \leq k \leq n} = (s^k, (\text{parameter}_1^k), (\text{bd}_1^k), \text{continuation}^k)_{1 \leq k \leq n}$ of partial interactions, such that:

1. $\forall k \in \llbracket 1; n \rrbracket, \text{exhibits}(t^k, \text{pi}^k)$;
2. $\forall k \in \llbracket 1; n \rrbracket, \text{components}(k) = s^k$;
3. $\text{sync}_{\text{part}}((t^k), (\text{parameter}_1^k), \text{compatibility}) \neq \perp$.

Then

$$(c_b, \text{part}) \xrightarrow{(\alpha^i)^n} (c_b \cup \text{new_behaviors}, \text{part} \setminus_{\text{part}} \text{removed} \cup_{\text{part}} \text{new_threads})$$

with:

- $\text{removed} = \text{remove}_{\text{part}}((t^k), (s^k))$;
- $(\text{new_behaviors}, \text{new_threads_part}) = \bigcup_{1 \leq k \leq n} \text{launch}_{\text{part}}(\text{Ct}^k, \bar{\text{id}}, \bar{E}^k)$,
where $\forall k \in \llbracket 1; 3 \rrbracket$:
 - $\text{Ct}^k \in \text{continuation}^k$;
 - $\bar{\text{id}} = \text{marker}((p^{k'}, \text{id}^{k'}, E^{k'})_{1 \leq k' \leq n})$;
 - $\bar{E}^k = \text{vpassing}(k, (t^{k'})_{1 \leq k' \leq n}, ((\text{bd}_1)^k)_k, ((\text{parameter}_1)^k)_k, \text{communications})$.
- $\forall k \in \llbracket 1; n \rrbracket, \alpha^k = (\mathcal{R}, (t^k), (\text{pi}^k))$.

where \cup_{part} and \setminus_{part} denote the usual sets join and sets minus operators respectively point-wise extended to the set $\mathcal{C}_{\text{part}}$ indexed by address values.

Soundness

We now state the soundness of this partitioned version of the non standard semantics for CAP.

Theorem 4.5 (Equivalence) *The partitioned collecting non standard semantics and the collecting non standard semantics are in strong bisimulation.*

Proof 4.6 *One can exhibit a Galois bijection between the \mathcal{C} and $\mathcal{C}_{\text{part}}$. A Galois bijection is a Galois connection (α, γ) where both α and γ functions are bijection. Such connection does not introduce any abstraction. It is also called an exact abstraction.*

The α function maps a partitioned non standard configuration to a non standard configuration:

$$\alpha(\mathbf{u}, (\mathbf{c}_{\text{beh}}, \text{part})) = (\mathbf{u}, \mathbf{c}_{\text{beh}} \cup \{\text{part}(\text{address}) \mid \text{part}(\text{address}) \text{ is defined}\})$$

The γ function is more complex, it partitions the set of threads of given non standard configuration into a set of partition units.

$$\gamma(\mathbf{C}) = (\mathbf{c}_{\text{beh}}, \text{part})$$

where $\mathbf{C} = \mathbf{c}_{\text{beh}} \cup \{\text{part}(x) \mid \text{part}(x) \text{ is defined}\}$ is a partition of \mathbf{C} .

Furthermore

$$\forall t = (p, \text{id}, E), \begin{cases} t \in \mathbf{c}_{\text{beh}} & \implies s_t = \{\text{behavior}_n\} \\ t \in \text{part}(x) & \implies \text{address}(t) = x \end{cases}$$

$$\text{where } s_t = \left\{ s \mid \begin{array}{l} \exists (s, \text{param}, \text{bounded}, \text{cont}) \in \text{partial_interactions}, \\ \text{exhibits}(p, \text{pi}) \end{array} \right\}.$$

This Galois bijection is compatible with both semantics. Therefore they are in strong bisimulation. This relation also applies for the collecting semantics lifting of both the non standard semantics and partitioned non standard semantics.

Example

Let us now illustrate this partitioning on an example that we follow all along this chapter. The example term given in Example 4.1 describes a simple system. Initially a single actor is associated to address a . It is able to handle messages labeled `install`. Two messages with such labels are sent to this address a . When the actor receives such a message, it dies but associates its behavior to the argument variable x of the received message. It also produces a new `install` labeled message containing a new value b .

There is therefore two different initial traces. On the one hand, the actor can receive the two initial messages, one after the other, replicating on address a . Then it handles one of the two produced `install` messages. It is, at that

stage, in a state where it can handle the last produced install message and be installed on this last address, recursively. The other first produced install message remains in the configuration forever.

On the other hand, the initial trace is the handling of only one of the two initial message, followed by an infinite sequence handling produced messages on addresses b. In that case, the remaining initial message sent on address a remains in reachable configurations.

Example 4.1 CAP example for the thread partitioning.

$$\begin{aligned} & \forall a^\alpha, \quad a \triangleright^1 [\text{install}^2(x) = \zeta(e, s) (x \triangleright^3 s \parallel \forall b^\beta, x \triangleleft^4 \text{install}(b))] \\ & \parallel \quad a \triangleleft^5 \text{install}(a) \parallel a \triangleleft^6 \text{install}(a) \end{aligned}$$

The Figure 4.2 illustrates one of these initial transitions in the partitioned flavor of the non standard semantics.

4.1.2 Abstract address partitioning

We now consider the abstraction of the partitioned concrete semantics.

Merging recursive instance

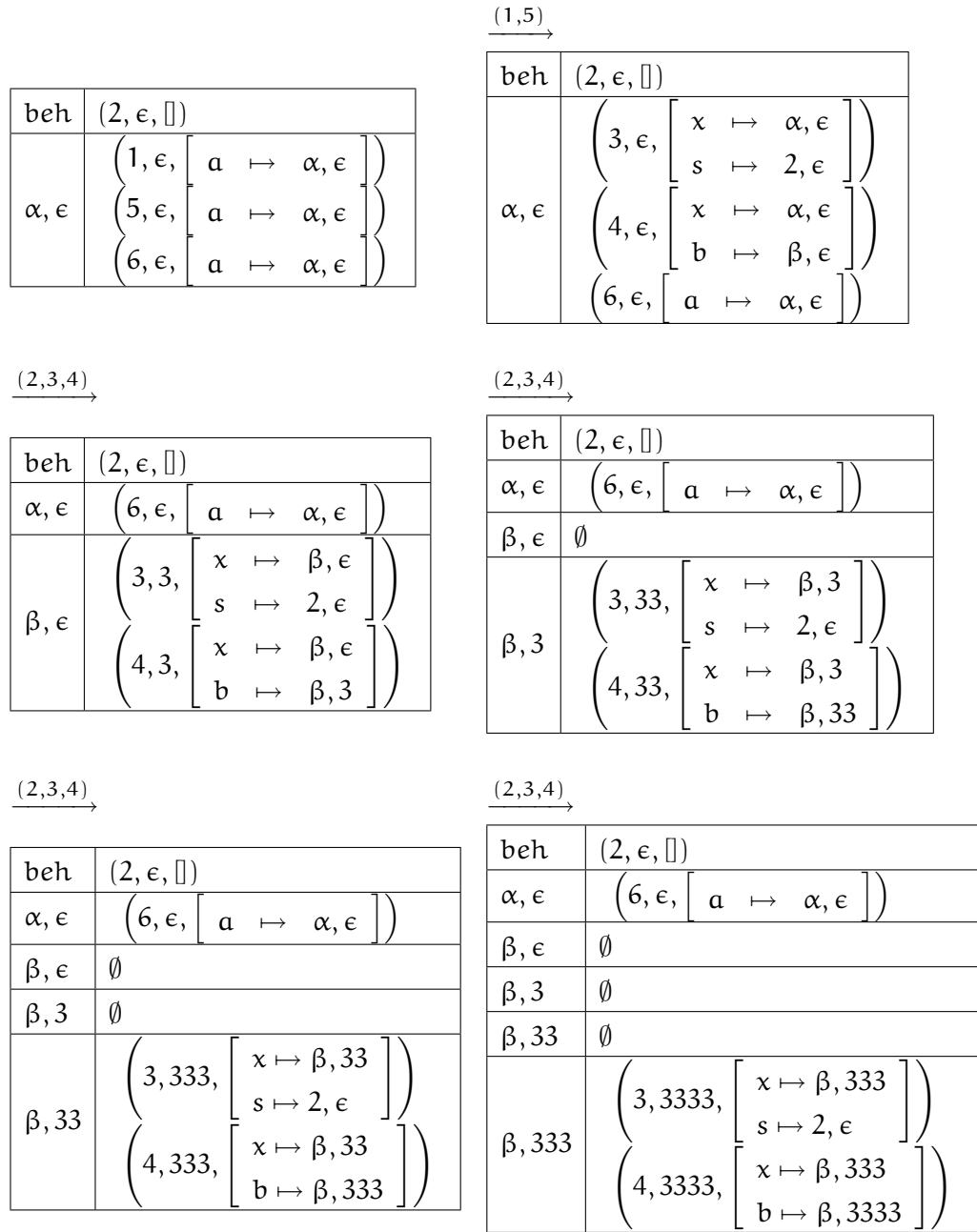
A first need when defining an abstraction is to obtain a decidable one. A drawback of current formalization of the partitioned concrete non standard semantics lies in the possibly unbounded number of partition units.

The solution proposed is to merge abstract properties specific to a given binder. The abstract element of a partitioned abstract domain is then a pair $(b, X^\#)$ where b denotes the binder and $X^\#$ the properties specific to this binder, expressed in an underlying abstract domain.

The abstract continuation launching takes care of safely merging partition elements associated to recursive instances of the same binder. When launching actor or message threads that are bound to a new address, they are launched in a new abstract element of the underlying domain. The union computation in the fixed point iteration merges both abstract elements to obtain properties that are valid for both the previously seen instances of the binder and the new one.

The choice of partitioning abstract threads, *i.e.* properties, by binders instead of addresses gives a decidable abstraction and allows to represent properties valid for all addresses on this binder.

In fact the partitioning could be defined with any abstraction of the address value. For example, one could imagine to partition abstract threads by both the binder and an abstract representation of its marker. This could give more precise information while still lying in a decidable framework.

Figure 4.2 CAP example concrete address partitioning.

In this example trace, the threads are partitioned depending on their associated address. The single behavior definition ever produced is kept in a special partition unit beh. This system is non-terminating and contains a bounded number of threads. However it creates infinitely many partition units, as it associates a new actor and a new message on a new address at each message reception.

Parametric underlying domain

A partitioned abstract domain has to rely on two abstractions. A first one is a control flow abstraction. It allows to over-approximate the address and therefore the binder of each launched thread.

The second abstraction is associated to each partition unit, approximating the set of actors and messages associated to the given binder.

We now impose a set of primitives and their soundness requirements that underlying domains must define and satisfy. They are then used in the partitioned abstract operational semantics to compute abstract transitions on the set of partition units.

Definition 4.7 (Underlying abstract domain) *Let $(\mathcal{C}^{\text{base}}, \sqsubseteq^{\text{base}}, \sqcup^{\text{base}}, \perp^{\text{base}}, \gamma_{\text{base}}, \mathcal{C}_0^{\text{base}}, \nabla^{\text{base}})$ be a sound abstract domain with respect to the condition (1-5,7) of Definition 3.3. This domain must also specify the following primitives:*

- A synchronization primitive $\text{sync}^{\text{base}}$ such that:
for all $x \in \mathcal{C}^{\text{base}}$ and transition involving the rule \mathcal{R} on the program points $(p^k)_k$ and the parameters $(\text{parameter}^k)_k$:

$$\left\{ (u, C) \in \gamma_{\text{base}}(x) \mid \begin{array}{l} \forall k, (p^k, \text{id}^k, E^k) \in C \text{ and} \\ \exists C' \text{ s.t. } C \xrightarrow{\mathcal{R}, (p^k)} C' \end{array} \right\} \\ \subseteq \gamma_{\text{base}}(\text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), x))$$

- A primitive computing the transition on the current interacting partition unit: update. It must satisfy:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists \text{id} \in \mathcal{M}, (u, C_{|(a, \text{id})}) \subseteq \gamma_{\text{base}}(\text{unit}_a) \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|(a, \text{id})} \cap \gamma_{\text{base}}(\text{synced}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ C'_{|(a, \text{id})} = C_{|(a, \text{id})} \setminus \left\{ \begin{array}{l} (p^a, \text{id}^a, E^a); \\ (p^m, \text{id}^m, E^m) \end{array} \right\} \cup \text{launch}(\text{cont}_a) \end{array} \right\} \right\} \\ \subseteq \gamma_{\text{base}}(\text{update}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}, a, \text{unit}_a, \text{cont}_a))$$

where $C_{|x}$ denotes the subset of threads in C associated to the address x and launch denotes the concrete computation of threads launching. \mathcal{R} denotes a formal rule, on the program points $(p^k)_k$ and partial interactions $(pi^k)_k$, considering the abstract synchronized element $\text{synced} \in \mathcal{C}^{\text{base}}$, the current partition unit binder $a \in \mathcal{L}_v$ and its associated interacting abstract element $\text{unit}_a \in \mathcal{C}^{\text{base}}$, and the set of static threads to be launched $\text{cont}_a \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$.

- A launch primitive *launch* that launches continuations considering the given synchronized abstract element *synced*. It must satisfy:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_V \times \mathcal{M}, \text{ s.t.} \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|x} \cap \gamma_{\text{base}}(\text{synced}) \\ \forall \text{id} \in \mathcal{M}, (u, C_{|(b, \text{id})}) \subseteq \gamma_{\text{base}}(\text{unit}_b) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ \exists \text{id}' \in \mathcal{M}, C'_{|(b, \text{id}')} = C_{|(b, \text{id}')} \cup \text{launch}(\text{cont}_b) \end{array} \right\} \right\} \\ \subseteq \gamma_{\text{base}}(\text{launch}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}, b, \text{unit}_b, \text{cont}_b)),$$

where $C_{|x}$ denotes the subset of threads in C associated to the address x and launch denotes the concrete computation of threads launching. \mathcal{R} denotes a formal rule, on the program points $(p^k)_k$ and partial interactions $(pi^k)_k$, considering the abstract synchronized element $\text{synced} \in \mathcal{C}^{\text{base}}$, the current partition unit binder $b \in \mathcal{L}_V$ and its associated abstract element $\text{unit}_b \in \mathcal{C}^{\text{base}}$ and the set of static threads to launch $\text{cont}_b \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$.

- A primitive to launch behavior threads, *launch_beh*, such that:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_V \times \mathcal{M}, \text{ s.t.} \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|x} \cap \gamma_{\text{base}}(\text{synced}) \\ C_{|\text{beh}} \subseteq \gamma_{\text{base}}(\text{beh}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ C'_{|\text{beh}} = C_{|\text{beh}} \cup \text{launch}(\text{cont}_{\text{beh}}) \end{array} \right\} \right\} \\ \subseteq \gamma_{\text{base}}(\text{launch_beh}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}, \text{beh}, \text{cont}_{\text{beh}})),$$

using similar arguments as above.

4.2 PARAMETRIC ABSTRACT PARTITIONING

4.2.1 Intuition

As mentioned above, the partitioned abstract domain we propose relies on two underlying domains.

The first domain approximates control flow. As before, it drives the abstract transitions, forbidding some spurious ones.

Then, relying on the abstract element associated to each interacting actor and message program point, we compute the set of partition units, where both actor and message exist.

Once the transition is computable, we rely again on the computed control flow information to project the launched threads among the different partition units. It is not a partition *sensu stricto* of the continuation, as each thread can be associated to different partitions depending on the precision of the control flow approximation.

For each possible interacting actor and message unit that can compute the transition, each unit is then updated, launching new threads associated to it, with the synchronization element obtained at the previous step for the considered unit.

We first introduce the parametric abstract domain. Then we define its abstract operational semantics.

4.2.2 Abstract Domain

Lattices

We first consider the control flow abstract domain $(\mathcal{C}^{env}, \sqsubseteq^{env}, \sqcup^{env}, \perp^{env}, \gamma_{env}, C_0^{env}, \rightarrow^{env}, \nabla^{env})$ as presented in Chapter 3. This domain drives the abstract transitions and provides an over-approximation of binders associated to threads in order to soundly represent them in the different partition units.

Then we consider the abstract domain $(\mathcal{C}^{base}, \sqsubseteq^{base}, \sqcup^{base}, \perp^{base}, \gamma_{base}, C_0^{base}, \nabla^{base})$ as a partitioned domain satisfying the Definition 4.7. Elements of \mathcal{C}^{base} are associated to each partition unit labels.

We introduce Units as such a finite set of partition unit labels. We define Units as the set of binder program points: $Units = \mathcal{L}_v$.

The partitioned part of our domain can then be defined as:

$$\text{Part}(Units, \mathcal{C}^{base}) = \mathcal{C}^{base} \times (Units \times \mathcal{C}^{base}).$$

We extend the pre-order relation point-wise, the bottom element and the union operator of \mathcal{C}^{base} to pairs $(u, c^\#) \in \text{Part}(Units, \mathcal{C}^{base})$. Then the pre-order \sqsubseteq^{part} , the union operator \sqcup^{part} , the bottom element \perp^{part} and the widening operator ∇^{part} are defined component-wise using either their equivalent in \mathcal{C}^{base} or their point-wise extension to $(Units \times \mathcal{C}^{base})$.

The main domain is now a product of the control flow abstraction and the partitioned one.

$$\mathcal{C}^\# \triangleq \mathcal{C}^{env} \times \text{Part}(Units, \mathcal{C}^{base})$$

All operators of the considered principal domain are defined pair-wisely. It defines the following lattice:

$$(\mathcal{C}^\#, \sqsubseteq^\#, \sqcup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightarrow^\#, \nabla^\#)$$

Concretization

Let us consider the concretization operators associated to the different abstract domains embedded in our partitioned abstraction.

The main concretization $\gamma^\#$ is defined as in Cartesian products:

$$\gamma^\#(cf_flow, part) \triangleq \gamma_{env}(cf_flow) \cap \gamma_{part}(part)$$

The concretization γ_{part} of the partitioned part ensures that any thread in any reachable configuration is included in the concretization γ_{base} of the abstract element associated to its address, or the behavior abstract element when considering behavior threads.

$$\left\{ (u, C) \left| \begin{array}{l} \forall (p, \text{id}, E) \in C, \\ \text{either } p \in \mathcal{L}_b \text{ and } (p, \text{id}, E) \in \gamma_{\text{base}}(\text{beh}) \\ \text{either } \left\{ \begin{array}{l} p \in \mathcal{L}_a \cup \mathcal{L}_m, \\ \exists \text{id}' \in \mathcal{M}, E[\text{address_var}(p)] = (a, \text{id}'), \\ \exists (a, \text{unit}_a) \in \text{cu}, (p, \text{id}, E) \in \gamma_{\text{base}}(\text{unit}_a) \end{array} \right. \end{array} \right. \right\} \subseteq \gamma_{\text{part}}(\text{beh}, \text{cu})$$

4.2.3 Semantics primitives

In order to define the initial abstraction $C_0^\#$ of our principal domain as well as its associated abstract transition $\rightarrow^\#$, we first introduce some primitives.

Unit label over-approximation

An essential part of this partitioned abstraction is the sound over-approximation of the addresses associated to threads. The primitive $\text{address}^\#$ computes such approximation, relying on an abstract control flow element.

ADDRESS BINDER APPROXIMATION The primitive $\text{address}^\# : \mathcal{L}_p \times \mathcal{C}^{\text{env}} \rightarrow \mathcal{L}_p$ is the abstract counterpart of the address primitive presented above in Section 4.1.1. It maps a thread program point to its possible binders considering an abstract control flow element.

Definition 4.8 (thread address binder) Let $p \in \mathcal{L}_p$ be a program point and $c_{\text{env}} \in \mathcal{C}^{\text{env}}$ an abstract element for the control flow approximation. We define the primitive $\text{address}^\#(p, c_{\text{env}})$ as:

$$\text{address}^\#(p, c_{\text{env}}) \triangleq \left\{ b \left| \begin{array}{l} (p, \text{id}, E) \in \gamma_{\text{env}}(c_{\text{env}}) \\ \exists b \in \mathcal{L}_v, m \in \mathcal{M}, \text{ s.t. } (b, m) = \text{address}((p, \text{id}, E)) \end{array} \right. \right\}$$

In practice, abstract domains used in the control flow part can be extended to construct such information. For example, the shape approximation, presented in Section 3.3.2, could be easily extended in such a way, returning initial nodes of the program point address variable.

THREAD ADDRESS VARIABLE Another simple but helpful primitive address_var associates to each valid program point its environment variable that describes its address.

Definition 4.9 (thread address variable) Let $p \in \mathcal{L}_p$ be an actor or message program point. Let $\pi_i = (s, (\text{parameter}_i), (\text{bd}_i), \text{continuation})$ be a partial interaction such that $\text{exhibits}(p, \pi_i)$.

We define the function $\text{address_var}(p)$. It relies on term extraction to identify the address variable of a given thread.

$$\text{address_var}(p) \triangleq \text{parameter}_1 \quad \text{when } s \in \left\{ \begin{array}{l} \text{static_actor}_n, \\ \text{dynamic_actor}, \\ \text{message}_n \end{array} \right\}.$$

Local synchronization

The local synchronization is computed in multiple steps and necessitates some specific primitives. We first sketch the synchronization, introducing new primitives. Their definition follows.

1. Relying on the control flow abstract element, the synchronization is enforced using the reagents[#] primitive of the underlying control flow abstract domain.
2. Using this control flow abstract element, binders that can be associated to both address of interacting actor and message threads are extracted (primitive $\text{address}^{\text{env}}$).
3. The synchronization constraints are enforced in the underlying partitioned abstract element associated to partition units associated to these extracted possible binders (primitive $\text{sync}^{\text{part}}$).
4. The abstract transition is computable if the synchronized control flow abstract element is not bottom and if at least one synchronized partitioned abstract element is not bottom.

SYNCHRONIZING PARTITIONED ABSTRACT ELEMENT The synchronization computation in the partitioned $\mathcal{C}^{\text{base}}$ domain has two purposes. The first one is to restrict the transition. The second one is to keep this modified abstraction and rely on it when launching new threads in an existing unit or in a new one.

Definition 4.10 (partitioned synchronization) Let \mathcal{R} be a formal rule. Let $(p^k)_k \in \mathcal{L}_p^k$ be a set of interacting thread program points. Let $(\text{parameter}^k)_k$ be a k -tuple of parameter variables. Let $\text{units} \in \text{Units}$ be a set of partition unit labels and $\text{part} \in \text{Part}(\text{Units}, \mathcal{C}^{\text{base}})^{\#}$ be an abstract partitioned element, a set of pairs, associating abstract elements in $\mathcal{C}^{\text{base}}$ to partition unit labels in Unit . We introduce the primitive $\text{sync}^{\text{part}}$ that ensures synchronization constraints in the partitioned part of the abstract domain element (beh, cu) :

$$\text{sync}^{\text{part}}(\mathcal{R}, (p^k)_k, \text{units}, (\text{parameter}^k), \text{units}, (\text{beh}, \text{cu})) \triangleq \left. \begin{array}{l} \mathbf{u} \in \text{units}, \\ (\mathbf{u}, c_{\text{part}}) \in \text{cu} \\ (u, c''_{\text{part}}) \left| \begin{array}{l} c'_{\text{part}} = \begin{cases} c_{\text{part}} & \text{when } \mathcal{R} = \text{static_trans}_k \\ \text{beh} \sqcup^{\text{base}} c_{\text{part}} & \text{otherwise} \end{cases} \\ c''_{\text{part}} = \text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), c'_{\text{part}}), \\ c''_{\text{part}} \neq \perp_{\text{base}} \end{array} \right\} \end{array} \right\}.$$

Launching continuations

We now update the current abstract element to the continuation launching. The control flow abstract element is independent and does not depend on the partitioned part. The resulting control flow element is then computed using the abstract transition of the underlying control flow abstract domain.

The thread launching in the partitioned part is more complex. Let us first enumerate the important points and then introduce the primitives that support this continuation launching.

- The abstract synchronization stage has computed an over-approximation of possible units containing interacting threads. The continuation launching has then to be computed for each of these units.
- Considering a set of static threads to be launched, a similar approach applies. Each possible partition unit label is mapped to the subset of the continuations that may be associated to this unit label. Then the resulting partition unit is updated considering (1) a synchronization unit (2) its associated subset of continuations.
- A special care is given to the continuation threads which denote actors or messages associated to a newly created address. In that case, we rely on a specific launching primitive, that allows to safely deal with the merge of recursive instances.

CONTINUATION THREADS BY UNIT As presented above, it is necessary to project continuation threads among the different unit labels. We introduce two primitives: `conts_of_units` and `new_units` that construct the set of threads potentially associated to a unit label, within an old abstraction or in a new one, respectively. We also define a simple primitive that extracts behavior threads from a continuation.

Splitting continuations associated to already created values Static threads in continuations have to be split according to their associated address. The

control flow part gives us valuable information concerning the value associated to these static threads when they are launched.

The continuation splitting is done in two parts. The first one identifies the static threads that could be associated to a given binder. Each of these binder specific continuations has again to be split according to equality and inequality relations of the control flow abstract element.

Definition 4.11 (continuation splitting by binder) *Let $(\text{cont}^k)_k$ be a k -tuple of continuations and $\text{cf} \in \mathcal{C}^{\text{env}}$ be an abstract control flow element. We introduce the primitive $\text{conts_of_units_by_binder}$ that builds the set of possible attributions of subset of continuations to a partition unit label. It does not consider continuation static thread where the address variable (let say v), contains a new value ($E_s[v]$ is defined).*

$$\text{conts_of_units_by_binder}((\text{cont}^k)_k, \text{cf}) \triangleq \left\{ (u, \text{cont}_u) \left| \begin{array}{l} (p, E_s) \in (\text{cont}^k)_k \\ \wedge u \in \text{address}^\#(p, \text{cf}) \\ \wedge \text{address_var}(p) \notin E_s \end{array} \right. \implies (p, E_s) \in \text{cont}_u \right\}$$

We now build the set of possible partitions of continuation identifying possible different markers. We introduce an equivalence relation built on static threads. We first define equalities among thread address variables and then use them to combine possibly equal equivalence classes.

Definition 4.12 (Equality transitive closure) *We define the \sim_{mol} as the equivalence relation between static threads (p_1, E_{s1}) and (p_2, E_{s2}) considering a control flow element molecule $\text{mol} \in \text{Mol}^\#$ such that*

$$(p_1, E_{s1}) \sim_{\text{mol}} (p_2, E_{s2}) \text{ iff } \{(\text{address_var}(p_1) = \text{address_var}(p_2)) \in \text{mol}\}$$

We denote by $\text{conts}_{\sim_{\text{mol}}}$ the set of equivalence classes $[(p, E_s)]_{\sim_{\text{mol}}}$ of conts .

Definition 4.13 (continuation splitting by address values) *We define the set of possible continuations conts_by_marker built on the continuation $\text{conts} \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ considering the abstract control flow molecule $\text{mol} \in \text{Mol}^\#$ such that:*

$$\text{conts_by_marker}(\text{conts}, \text{mol}) \triangleq \left\{ \text{cont} \left| \begin{array}{l} \exists c_1, \dots, c_n \in \text{conts}_{\sim_{\text{mol}}} \text{ s.t } \text{cont} = c_1 \cup \dots \cup c_n \\ \text{and } \forall c_i \neq c_j \in \text{cont}, \\ \neg((\text{address_var}(c_i) \neq \text{address_var}(c_j)) \in \text{mol}) \end{array} \right. \right\}$$

Definition 4.14 (continuation splitting) *Let $(\text{cont}^k)_k$ be a k -tuple of continuations and $\text{cf} \in \mathcal{C}^{\text{env}}$ be an abstract control flow element. Let mol be the abstract molecule obtained when computing the synchronization on the abstract control flow*

element. We introduce the primitive `conts_of_units` that builds the set of possible attributions of subset of continuations to a partition unit label. It does not consider continuation static threads where the address variable (let say v), contains a new value ($E_s[v]$ is defined).

$$\text{conts_of_units}((\text{cont}^k)_k, \text{cf}, \text{mol}) \triangleq \left\{ (u, \text{conts}) \mid \begin{array}{l} (u, \text{cont}) \in \text{conts_of_units_by_binder}((\text{cont}^k)_k, \text{cf}) \\ \text{and } \text{conts} = \text{conts_by_marker}(\text{cont}, \text{mol}) \end{array} \right\}$$

Splitting continuations associated to newly created values Newly created values are more easily identified. A single new value is created for each v binder. There is no possible way to unsafely identify a newly created value with other – old or new – ones.

Definition 4.15 (new address in continuations) Let $(\text{continuation}^k)_k$ be a k -tuple of continuations and $\text{cf} \in \mathcal{C}^{\text{env}}$ be an abstract control flow element. We now introduce the primitive `new_units`. Similarly to the `conts_of_units` primitives, it extracts the continuation static thread that is associated to new addresses during the current transition computation. The continuations are associated to a single address binder in the `new_units` primitives, and are therefore continuations, i.e. in $\wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$, instead of non deterministic continuations, i.e. $\wp\wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$, like in the preceding primitive.

$$\text{new_units}((\text{continuation}^k)_k, \text{cf}) \triangleq \left\{ (u, \text{conts}) \mid \begin{array}{l} (p, E_s) \in (\text{continuation}^k)_k \\ \wedge u \in \text{address}(p, \text{cf}) \\ \wedge \text{address_var}(p) \in E_s \end{array} \implies (p, E_s) \in \text{conts} \right\}$$

Behavior threads in continuations

Definition 4.16 (Extracting behaviors) Let $\text{conts} \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}))$ be a set of static threads. We define the primitive `beh_conts` that extracts behavior static threads from this continuation:

$$\text{beh_conts}(\text{conts}) \triangleq \{(p, E_s) \in \text{conts} \mid p \in \mathcal{L}_b\}$$

LAUNCHING THREADS Again the launching is split in three primitives. A first updates the interacting unit, removing consumed threads and launching new ones. A second considers the update of an existing partitioned abstract element, when the third rather builds a new partition unit, which is merged with existing ones later during the union computation at the end of the abstract transition.

Definition 4.17 (updating interacting unit) Let \mathcal{R} be a formal rule. Let $(p^k)_k \in \mathcal{L}_p^k$ be a set of interacting thread program points and $(pi^k)_k$ their associated partial interactions. Let $unit_u \in \mathcal{C}^{base}$ be an abstract element denoting the interacting thread unit and let $synced \in \mathcal{C}^{base}$ be its synchronization. Let $conts_u \in \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}_v))$ be a set of non deterministic continuations to be launched.

We introduce the primitive $update^{part}$ as:

$$update^{part}(\mathcal{R}, (p^k), (pi^k), synced, unit_u, conts_u) \triangleq \bigsqcup_{cont \in conts_u}^{base} update^{base}(\mathcal{R}, (p^k), (pi^k), synced, unit_u, cont)$$

Definition 4.18 (launching continuations) Let \mathcal{R} be a formal rule. Let $(p^k)_k \in \mathcal{L}_p^k$ be a set of interacting thread program points and $(pi^k)_k$ their associated partial interactions. Let $synced \in \mathcal{C}^{base}$ be an abstract element denoting the synchronization of the interacting thread abstractions. Let $cu \in \text{Units} \times \mathcal{C}^{base}$ be the set of abstract partition units. Finally, let $conts \in \text{Units} \times \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}_v))$ be a set of non deterministic continuations to be launched, separated into subsets associated to their possible partition unit label.

We introduce the primitive $launch^{part}$ as:

$$launch^{part}(\mathcal{R}, (p^k), (pi^k), synced, cu, conts) \triangleq \bigsqcup_{cont \in conts_u}^{base} (u, launch^{base}(\mathcal{R}, (p^k), (pi^k), synced, u, unit_u, conts))$$

such that $\left\{ \begin{array}{l} (u, conts_u) \in conts \\ (u, unit_u) \in cu \end{array} \right.$.

Definition 4.19 (launching fresh unit) Let \mathcal{R} be a formal rule. Let $(p^k)_k \in \mathcal{L}_p^k$ be a set of interacting thread program points and $(pi^k)_k$ their associated partial interactions. Let $synced$ be an abstract element denoting the synchronization of the interacting threads abstractions. And let $conts \in \text{Units} \times \wp(\mathcal{L}_p \times (\mathcal{V} \rightarrow \mathcal{L}_v))$ be a set of continuations to be launched on fresh addresses, separated into subsets associated to their possible partition unit label.

We introduce the primitive new_launch which carefully deal with recursive instances of a value:

$$new_launch^{part}(\mathcal{R}, (p^k), (pi^k), synced, conts) = \left\{ \begin{array}{l} (u, launch^{base}(\mathcal{R}, (p^k), (pi^k), synced, u, \perp^{base}, conts_u)) \mid \\ (u, conts_u) \in conts \end{array} \right\}$$

4.2.4 Operational semantics

Now we rely on the above primitives as well as the underlying abstract domain primitives given in Definition 4.7 to describe the initial abstract element and the abstract transition rule.

The initial state $C_0^\#$ is obtained from the set of initial static threads init_s and is defined as

$$(C_0^{\text{env}}, (C_0^{\text{base}}(\text{init_beh}), C_0^{\text{part}}))$$

where

- C_0^{env} is the initial abstract element of the underlying control flow abstraction considering the initial set init_s of static threads;
- $C_0^{\text{base}}(S)$ denotes the initial abstract element of the underlying partitioned abstract domain considering the initial set S of static threads;
- $\text{init_beh} \subset \text{init}_s$ is the subset of initial threads that denote behavior threads;
- C_0^{part} is defined as

$$\left\{ (u, C_0^{\text{base}}(\text{conts})) \mid \begin{array}{l} \forall (p, E_s) \in \text{init}_s, \\ E_s[\text{address_var}(p)] = u \implies (p, E_s) \in \text{conts} \end{array} \right\}.$$

The partitioned abstract domain semantics relies on the abstract primitives defined above and is described in Figure 4.3. We recall informally the different steps of the abstract transition computation:

- find a transition rule and a tuple of interacting threads, exhibiting the appropriate partial interactions with respect to the rule;
- compute the set of units that can be associated to both the actor and the message threads, using the control flow part of the abstract element;
- restrict the set of considered units to the ones where the abstract element associated to these units satisfy the synchronization constraints defined by both the actor and the message threads;
- for each of these valid interacting units, launch threads in continuation, if the transition is computable:
 - compute, using the control flow part, the set of static threads, subset of the continuations, that can be launched in the different partition units;
 - Apply the launch primitive of the underlying partitioned domain to each unit considering the subset of launched threads for the unit.

Theorem 4.20 *The abstraction $(\mathcal{C}^\#, \sqsubseteq^\#, \sqcup^\#, \perp^\#, \gamma^\#, C_0^\#, \rightarrow^\#, \nabla^\#)$ is a sound abstraction with respect to the Definition 3.3 considering a sound underlying control flow abstraction and an abstract domain $\mathcal{C}^{\text{base}}$ satisfying the soundness assumption of Definition 4.7.*

Proof 4.21 *The proof can be found in A.2.*

Figure 4.3 Partitioned abstract domain operational semantics.

Let $C^\# = (cf, (beh, cu) \in \mathcal{C}^\#$ be an abstract configuration where $cf \in \mathcal{C}^{env}$ denotes the control flow abstract element and $(beh, cu) \in \text{Part}(\text{Units}, \mathcal{C}^{base})$ the partitioned element: $beh \in \mathcal{C}^{base}$ denotes the abstract element specific to behavior threads and cu , the computation units, is a set of pairs $(unit, X^\#)$ where $unit \in \mathcal{L}_v$ is a partition unit identifier and $X^\# \in \mathcal{C}^{base}$ is an abstract element of the underlying partitioned abstract domain.

Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_passing)$ be a formal rule. Let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program points and $(pi^k)_{1 \leq k \leq n} = (s^k, (\text{params}^k), (\text{bd}^k), \text{conts}^k)$ be the tuple of exhibited partial interactions. Let p^a and p^m in (p^k) be the actor and message program points respectively. We introduce

$$\begin{aligned} \text{units}_{p^k} &= \text{address}^{env}(p^k, cf) & \text{units} &= \text{units}_{p^a} \cap \text{units}_{p^m} \\ \text{mol} &= \text{sync}^{env}((p^k), (\text{params}^k), \text{conts}^k, cf) \\ \text{synced_part} &= \text{sync}^{part}(\mathcal{R}, (p^k), (\text{params}^k), \text{units}, (beh, cu)) \end{aligned}$$

$$\text{When} \left\{ \begin{array}{l} \forall k \in \llbracket 1, n \rrbracket, pi^k \in \text{interaction}(p^k) \\ \text{mol} \neq \perp \\ \text{synced_part} \neq \emptyset \end{array} \right.$$

Then $(cf, (beh, cu)) \xrightarrow{\mathcal{R}, (p^k)_k, (pi^k)_k} \#(cf', (beh', cu'))$

Where

1. $cf \xrightarrow{\mathcal{R}, (p^k)_k, (pi^k)_k}^{env} cf'$;
2. $\overline{\text{conts_of_units}} = \text{conts_of_units}((\text{conts}^k)_k, cf', \text{mol})$;
3. $\overline{\text{new_units}} = \text{new_units}((\text{conts}^k)_k, cf')$;
4. $\text{interacting_conts}(u) = \{\text{conts} \mid (u, \text{conts}) \in \overline{\text{conts_of_units}}\}$;
5. $beh' = (\bigsqcup_{(u, \text{synced}) \in \text{synced_part}}^{base} \{\text{launch_beh}^{base}(\mathcal{R}, (p^k), (pi^k), \text{synced}, beh, \text{beh_conts}(\text{conts}))\})$;
6. $\text{unit}_u = \{\text{unit} \mid (u, \text{unit}) \in cu\}$;
7. $\text{launched_units} = \bigsqcup_{(u, \text{synced}) \in \text{synced_part}}^{base} \left\{ \begin{array}{l} \text{update}^{part}(\mathcal{R}, (p^k), (pi^k), \text{synced}, \text{unit}_u, \text{interacting_conts}(u)), \\ \text{launch}^{part}(\mathcal{R}, (p^k), (pi^k), \text{synced}, cu, \overline{\text{conts_of_units}}), \\ \text{new_launch}^{part}(\mathcal{R}, (p^k), (pi^k), \text{synced}, \overline{\text{new_units}}) \end{array} \right\}$;
8. $cu' = cu \bigsqcup^{base} \text{launched_units}$.

4.3 EXAMPLE ANALYSIS

We consider again the example presented above in Example 4.1. We present the abstract properties computed for a partitioned flavor of the occurrence counting abstraction and the control flow abstraction.

We briefly explain the modification necessary to express such domains as underlying domains for our partitioned analysis. Then we state some abstract properties obtained.

Partitioning occurrence counting

The main occurrence counting abstract domain is extended to define the following primitives:

- The $\text{sync}^{\text{base}}$ primitive is defined using the $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}$ primitive. The (p^k) argument of the $\text{sync}^{\text{base}}$ function defining the constraint t of $\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}$:

$$\text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), \text{elem}) \triangleq \text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, \text{elem})$$

$$\text{where } t = \lambda x. \begin{cases} 1 & \text{when } x \in (p^k)_k \\ 0 & \text{otherwise} \end{cases}.$$

- The $\text{update}^{\text{base}}$ primitive exactly corresponds to the initial abstract transition computation:

$$\text{update}^{\text{base}}(\mathcal{R}, (p^k)_k, (pi^k), \text{synced}, u, \text{unit}_u, \text{cont}_s) \triangleq \text{synced} +^{\#} \text{Transition} +^{\#} \Sigma^{\#} \beta^{\#}(\text{cont}_s) -^{\#} \text{Consumed}$$

where

- $\text{Transition} = 1_{\mathcal{N}_{\mathcal{V}_c}}(\psi((p^k)_k));$
- $\text{Consumed} = \Sigma^{\#}(1_{\mathcal{N}_{\mathcal{V}_c}}(p^k))_{k \in [1, n] \wedge \text{type}(s_k) \neq \text{replication}}$

- The $\text{launch}^{\text{base}}$ primitive considers only thread launching:

$$\text{launch}^{\text{base}}(\mathcal{R}, (p^k)_k, (pi^k), \text{synced}, u, \text{unit}_u, \text{cont}_s) \triangleq \text{unit}_u +^{\#} \text{Transition} +^{\#} \Sigma^{\#} \beta^{\#}(\text{cont}_s)$$

where $\text{Transition} = 1_{\mathcal{N}_{\mathcal{V}_c}}(\psi((p^k)_k));$

- $C_0^{\text{base}}(\text{cont}_s)$ is defined easily from the initial element abstraction $C_0^{\mathcal{N}_{\mathcal{V}_c}}$:

$$C_0^{\text{base}}(\text{cont}_s) \triangleq \beta^{\#}(\text{cont}_s).$$

The Figure 4.4 illustrates the use of the partitioned occurrence counting domain on the Example 4.1.

Figure 4.4 Partitioned occurrence counting.

The partitioned occurrence counting domain gives the following constraints:

Unit	Abstract element (Karr)
α	$\{H1 + R1 = 1; H3 - R1 = 0\}$
β	$\{H3 - R1 = 0; H4 - R1 = 0\}$

When the non partitioned use of this domain gives:

$$\left\{ \begin{array}{ll} H5 + R1 = 1 & H3 - R1 = 0 \\ H1 + R1 = 1 & H6 + H5 + H4 = 2 \end{array} \right\}$$

In that example, both partitioned and non partitioned are meaningful. The use of the partitioning allows to represent properties specific to an address binder but it loses any relation between occurrences of threads in different units.

In the first case, we observe that considering threads on addresses bound by β , there is at most one message on 4 ($H4 = R1$ and $R1$ is not recursive).

In the second, we can get more global properties considering the total number of messages in reachable configurations: here 2.

Partitioning control flow data

Similarly, we extend the main control flow abstract domain to define the following primitives:

- The $\text{sync}^{\text{base}}$ primitive is defined using the $\text{reagents}^{\#}$ primitive:

$$\begin{aligned} \text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), \text{elem}) &\triangleq \\ \text{reagents}^{\#}((p^k), (\text{parameter}^k), \text{compatibility}, \text{elem}) \end{aligned}$$

where $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_{\text{passing}})$.

- The $\text{launch}^{\text{base}}$ primitive contains all the steps following the synchronization that are needed to launch threads:
 - compute value passing;
 - create a new marker if needed;
 - effectively launch continuation threads.

$$\begin{aligned} \text{launch}^{\text{base}}(\mathcal{R}, (p^k)_k, (pi^k), \text{synced}, u, \text{unit}_u, \text{conts}_u) &\triangleq \\ \text{unit}_u \sqcup^{\text{base}} \text{launch}^{\#}((p^k), \text{conts}_u, \text{mol}') \end{aligned}$$

where

- $\mathcal{R} = (\mathbf{n}, \text{components}, \text{compatibility}, \text{v_passing})$;
 - $(\text{pi}^k)_k = (\text{s}^k, (\text{par}^k), (\text{bd}^k), \text{cont}^k)_k$;
 - $\text{mol}' = \text{marker_value}((\text{p}^k), \text{synced}, (\text{bd}^k), (\text{par}^k), \text{v_passing})$.
- In control flow abstract domains, underlying atom domains are not relational between program point threads. Furthermore thread removing when computing abstract transition is not represented. Therefore the primitive $\text{update}^{\text{base}}$ can be mapped to the $\text{launch}^{\text{base}}$ function. In the first case, all abstract information is thread specific.

$$\text{update}^{\text{base}}(\mathcal{R}, (\text{p}^k), (\text{pi}^k), \text{synced}, \text{u}, \text{unit}_{\text{u}}, \text{cont}_{\text{s}_{\text{u}}}) \triangleq \text{launch}^{\text{base}}(\mathcal{R}, (\text{p}^k), (\text{pi}^k), \text{synced}, \text{u}, \text{unit}_{\text{u}}, \text{cont}_{\text{s}_{\text{u}}})$$

- $C_0^{\text{base}}(\text{cont}_{\text{s}})$ is defined easily from the initial element abstraction C_0^{env} :

$$C_0^{\text{base}}(\text{cont}_{\text{s}}) \triangleq \text{launch}^{\#}(\text{cont}_{\text{s}}, \epsilon_{\emptyset}^{\#}).$$

The Figure 4.5 illustrates the use of the partitioned control flow domain on the Example 4.1.

4.4 ENHANCEMENT

The presented partitioned abstract domain is very parametric and allows to represent properties specific to a given address binder. It is able to safely deal with recursive creations of addresses, merging into the same abstraction the properties verified by all instances of an address.

We now present an extension of the previous partitioned abstraction that allows more precise properties to be computed. We first present the extension. Then we argue the soundness of the modified partitioned abstract domain. Finally we give an idea of the necessity for such an extension.

4.4.1 Extending primitives

This extension intends to provide more control flow information about interacting threads, or any thread in the reachable configurations, when computing a transition locally in a unit within the underlying partitioned abstract domain.




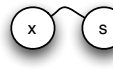

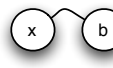




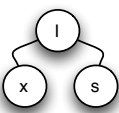


This idea is to provide an excerpt of the control flow abstraction computed to the underlying partitioned abstract semantic primitives.

This extension is defined in two steps:

1. We first modify the partitioned abstraction and some of its operational primitives to introduce control flow information available for the Base domain operational semantics primitives. Calls to these primitives use

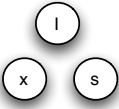

Figure 4.5 Partitioned control flow.

The partitioned control flow gives:

unit	Abstract element			
	p	Reg \mathcal{L}	Marker	Value
α	1	$I \mapsto \epsilon$ $a \mapsto \alpha.\epsilon$		
	3	$I \mapsto \epsilon$ $x \mapsto \alpha.\epsilon$ $s \mapsto 2.\epsilon$		
	4	$I \mapsto \epsilon$ $x \mapsto \alpha.\epsilon$ $b \mapsto \beta.\epsilon$		
	5	$I \mapsto \epsilon$ $a \mapsto \alpha.\epsilon$		
	6	$I \mapsto \epsilon$ $a \mapsto \alpha.\epsilon$		
	β	3	$I \mapsto 3^+$ $x \mapsto \beta.3^*$ $s \mapsto 2.\epsilon$	
4		$I \mapsto 3^+$ $x \mapsto \beta.3^*$ $b \mapsto \beta.3^*$		

We obtain that, considering addresses bound by α , the thread on program point 3 share the same marker value for the identity marker of the threads, their variable x and their variable s .

Considering threads associated to addresses bound by β , we have the disequality relation between the identity marker of threads and their variable x marker. When the global control flow abstraction gives for program point 3:

3	$I \mapsto 3^*$ $x \mapsto \alpha.\epsilon + \beta.3^*$ $s \mapsto 2.\epsilon$		
---	--	---	--

Both graph abstractions denote \top value and do not give any valuable information.

the \top^{CF} value of the control flow domain considered, contingent upon the Base domain needs, for the time being.

2. Then, each call to these modified primitives is preceded by a reduction computation between the local control flow element used and the global one existing in the control flow part of the partitioned abstract domain.

Enriching primitives

The idea is to instantiate the parametric part not only by Unit and $\mathcal{C}^{\text{base}}$ but with a third domain control flow abstraction $\text{CF} \in \mathcal{L}_p \rightarrow \text{Atoms}_{\mathbb{V}}^{\#}$. This CF domain is not fitted with an abstract transition relation, but it must be a base domain in our \mathcal{C}^{Env} abstraction. For example, the shape approximation presented in the previous chapter can be used.

The partitioned part is now defined as $\text{Part}(\text{Units}, \mathcal{C}^{\text{base}}, \text{CF})$. We now consider a modified version the $\mathcal{C}^{\text{base}}$ domain where the primitives $\text{sync}^{\text{base}}$, $\text{update}^{\text{base}}$, and $\text{launch}^{\text{base}}$ have the following signatures:

- $\text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), \text{elem}, \text{cf_elem});$
- $\text{update}^{\text{base}}(\mathcal{R}, (p^k)_k, (\text{pi}^k), \text{synced}, u, \text{unit}_u, \text{conts}_u, \text{cf_elem});$
- $\text{launch}^{\text{base}}(\mathcal{R}, (p^k)_k, (\text{pi}^k), \text{synced}, u, \text{unit}_u, \text{conts}_u, \text{cf_elem});$

where cf_elem denotes an element of CF.

Modifying calls

Now we modify the abstract transition in order to use these new primitives.

Let us first introduce the control flow element cf^{\top} as the abstract element that associates to each program point its top value in the considered control flow abstraction.

$$\text{cf}^{\top} = \lambda p. \top_{I(p)}^{\text{Atom}}$$

We recall that the control flow abstraction is non relational and approximates all possible pairs (id, E) such that it exists a thread (p, id, E) in reachable configurations.

For example, in the Shape case, the cf^{\top} element associates to each program point the element that gives for its marker and for each of its environment variables all the possible words in \mathcal{L}^* , denoted by the abstract value $\top_{\mathcal{L}}^{\text{Reg}}$.

We can then replace any call to the $\text{sync}^{\text{base}}$ or $\text{launch}^{\text{base}}$ with this cf^{\top} value for the variable cf_elem .

Computing reduction

The previous step allows to provide control flow information to underlying partitioned abstract domains but it gives a \top value, *i.e.* no valuable information.

We state that the control flow approximation used here in CF is also present in the global control flow approximation \mathcal{C}^{env} , as a base abstraction.

We define a simple reduction $\rho : \mathcal{C}^{env} \times CF$ on the pair $(cf, (beh, cu)) \in \text{Part}(\text{Units}, \mathcal{C}^{base}, CF)$. This reduction takes the element of the domain CF in cf and returns the intersection of both CF elements.

It is used in each of the primitives sync^{base} or launch^{base} restricting the cf_elem value to sound smaller abstraction.

For example, when computing the abstract transition

$$(cf, (beh, cu)) \xrightarrow{\mathcal{R}, (p^k)_k, (pi^k)_k} \#(cf', (beh', cu'))$$

any launched^{base} call is replaced by

$$\text{launch}^{base}(\mathcal{R}, (p^k)_k, (pi^k)_k, \text{conts}_u, \text{synced}, \text{initial}, \rho(cf, cf^\top)).$$

4.4.2 Soundness

Theorem 4.22 *The extended version of the partitioned abstract domain is a sound abstraction with respect to Definition 3.3.*

Proof 4.23 *The initial partitioned abstract is a sound abstraction with respect to Definition 3.3. Modifying the primitives to add a new parameter and calling them with a \top value for this parameter does not break the soundness of the domain.*

Then the reduction computation also preserves soundness, it restricts the value with the sound over-approximation computed in the global control flow part.

4.4.3 Application

The extended version of our partitioned domain can now be used on a wide variety of underlying abstract domains. These domains are intended to be used to abstract properties per address. But any more global abstraction such as the control flow one or the occurrence counting one could also be expressed as underlying domains of the partitioned abstraction.

We extend the primitives to provide control flow information to underlying abstractions in order to support thread launching. In fact, depending on the need of the underlying domain abstract semantics, one can need control flow information to launch threads more precisely.

This kind of use is exploited in the Chapter 7 where abstract properties represent causality, *i.e.* sequences of actor behaviors associated to an address. In that case, launching threads in the current unit is already precise: we know the last

actor installed, it is the interacting one. But when launching outside the interacting unit, a lot of imprecision arise that could be attenuated using the control flow abstract property already available.

One could also imagine underlying abstract domains which necessitate control flow information about interacting thread environments when computing the synchronization constraints.

4.5 RELATED WORK AND DISCUSSION

The presented domain partition abstracts properties per address binder. In that sense, it allows to represent properties in way similar to the properties usually expressed using type systems. In this other approach of static analysis, the term is analyzed once, associating a type to each sub-term. It only considers one instance of each name binder and could not easily differentiate their recursive instances.

In the presented abstraction, we are able to represent such properties while relying on the powerful framework of abstract interpretation. For example, our control flow approximation allows us to only consider a sound subset of possibly matched behavior branches while type system analysis faces more difficulties with respect to precision.

4.5.1 *Comparison with Feret's thread partitioning*

In [48] and in [49, Chap. 10], FERET proposed a thread partitioning analysis. This analysis is related to the one we presented here but differs in multiple points. It is first motivated by the analysis of Ambient calculi. In such calculi, communications are local to a place containing threads. All thread environments contain a variable *loc* expressing their container identity. A first difference is that such partitioning does not allow communications between multiple partition units. Any calculus that needs such a mechanism could not take benefit of this abstraction. In calculi that rely on a notion of definition threads available to every other thread these definitions have to be accumulated outside partition units and cannot be used in communications. CAP encoding or the Join calculus encoding face such a difficulty.

In its current definition, FERET's partitioning provides only a numerical abstraction for each partition unit. It is defined as a parametric domain instantiated with an occurrence counting abstraction and it relies directly on numerical primitives. Our proposal is, in that sense, more general, as it can be instantiated with any domain defining the appropriate primitives. In our context, this is also essential since we need more powerful abstractions than only occurrence counting ones when verifying CAP specific properties. The Chapter 7 introduces an abstract domain for analyzing the sequences of messages available to

an address. This domain is intended to be used under the partitioned abstract domain as it focuses on one address at a time. It carries causality constraints that could not be expressed only as numerical ones.

Finally, FERET's proposal is quite complex. It considers all the specificities of the non standard semantics that we do not mention in this thesis. For example, the full non standard semantics allows to express broadcasting mechanisms globally modifying thread environments. It is used to model a cell opening in Ambient. This leads to a complex abstraction which would be hardly provable in its generic formalization.

4.5.2 *Summary*

In this chapter, we proposed a parametric abstract domain that represents properties per address binder. All communications in the actor paradigm are address-centric. Such a partitioned domain is then essential to CAP specific properties analysis. It necessitates two domains. The first one approximates control flow information. It drives transitions and allows to soundly compute possible binders associated to interacting threads or launched ones. The second one carries the information associated to each binder.

This domain is a first step towards the analysis of high-level properties for CAP.

The goal of this chapter is to illustrate the use of the occurrence counting abstract domains in practice on examples. Then according to the expected results and the ones we obtain using these abstraction, we introduce three enhancements allowing to over-approximate more precisely the possible computations as well as the resulting abstract properties.

This chapter is sectioned in four parts. We first give, in Section 5.1, a description of the initial occurrence counting underlying abstract domains as presented by FERET in [49, Chap. 9]. We introduce the different domains and give the reduction algorithm between the relational and non relational abstract elements. We describe the reduction in an implementable way and give practical use of it on an example. The Section 5.2 proposes to extend the approximated reduction in order to handle more constraints and therefore obtain more precise results. The Section 5.3 introduces a domain that keeps track of the dependencies among computed transitions and threads needed for such transitions. It allows to forbid certain kinds of non computable transitions that were allowed in the preceding abstractions. Finally the last section, Section 5.4, gives a way to synchronize information between occurrence domains used under the partitioned abstract domain presented in the previous chapter, Chapter 4.

5.1 THE INITIAL OCCURRENCE COUNTING ABSTRACTION

We now give a description of the numerical abstract domains used to approximate occurrences of threads in reachable configurations as well as the over-approximation of computed transition labels.

As introduced in the previous chapter, in Section 3.3.3, each reachable configuration $(u, C) \in \Sigma^* \times \mathcal{C}$ is first abstracted by a family of natural numbers indexed by $\mathcal{V}_C = \mathcal{L}_p \times \mathbb{B}$ denoting the occurrence of threads in C indexed by program p $((p, T) \in \mathcal{V}_C)$ and the occurrence of computed transitions in u consuming the actor program point p $((p, F) \in \mathcal{V}_C)$. Each reachable configuration is then mapped to an element of $\mathbb{N}^{\mathcal{V}_C}$ ¹.

The set of reachable configurations $\{(u, C)\} \in \wp(\Sigma^* \times \mathcal{C})$ is then abstracted by another domain that represents sets of possible families of natural integers indexed by \mathcal{V}_C . Such elements can then over-approximate set of configurations using the combination of $\gamma_{\mathbb{N}^{\mathcal{V}_C}}$ by $\gamma_{\mathcal{N}^{\mathcal{V}_C}}$.

¹ We recall, that following the notation of FERET, $\mathbb{N}^{\mathcal{V}_C}$ denotes families of natural number indexed by \mathcal{V}_C

A numerical abstract domain used to represent sets of natural numbers families indexed by \mathcal{V}_c must satisfies the soundness assumptions of the Definition 3.7, page 63.

5.1.1 Numerical abstractions

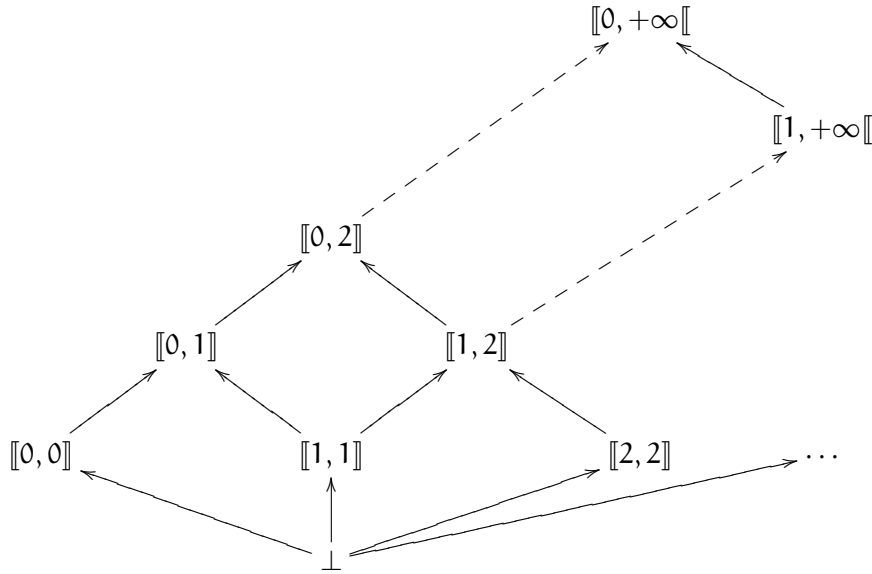
We now give the details of the two domains proposed in [49] as well as the way their primitives can be defined. We first introduce the non relational numerical abstraction based on the interval abstract domain. Then we describe the relational abstraction based on the domain proposed by KARR in [68], an abstract domain to model affine relationships between variables in \mathcal{V}_c . Finally we detail the reduction between these two domains.

Intervals

This first domain carries the essential information. It associates to each element of \mathcal{V}_c an interval denoting the over-approximation of thread occurrence or label occurrence in the transition words.

The principal primitives are defined component-wise. We recall that the intervals used are intervals of natural numbers in \mathbb{N} .

Figure 5.1 The lattice of intervals in \mathbb{N}^2 .



The abstract domain \mathcal{I} is associated to $\wp(\mathbb{N}^{\mathcal{V}_c})$ by the monotonic map $\gamma_{\mathcal{I}}$ defined as

$$\gamma_{\mathcal{I}}(f) = \{\mathbf{u} \in \mathbb{N}^{\mathcal{V}_c} \mid \forall i \in \mathcal{V}_c, u_i \in f(i)\}$$

- the union operator $\cup_{\mathcal{J}}$ is defined point-wisely using the union of natural integers:

$$f \cup_{\mathcal{J}} g = [x \mapsto f(x) \cup g(x)]$$

$$\text{with } \llbracket a; b \rrbracket \cup \llbracket c; d \rrbracket = \llbracket \min(a, c); \max(b, d) \rrbracket;$$

- the abstract domain of natural numbers intervals does not satisfy the ascending chain condition and allows infinite ascending chains. We introduce the widening operator $\nabla_{\mathcal{J}}^n$:

$$[f \nabla_{\mathcal{J}}^n g](x) = f(x) \nabla^n g(x)$$

$$\text{where } \llbracket a; b \rrbracket \nabla_{\mathcal{J}}^n \llbracket c; d \rrbracket =$$

$$\begin{cases} \llbracket \min(a, c); +\infty \rrbracket & \text{when } d > \max(b, n) \\ \llbracket \min(a, c); \max(b, d) \rrbracket & \text{otherwise} \end{cases}.$$

The integer n is a parameter of the analysis. In our case, we can take $n = 0$ since our main occurrence counting abstraction is a reduction between a non relational and a relational abstract domains. Then the reduction improves the precision of the value.

- the operator $+_{\mathcal{J}}$ is defined as the addition of the intervals associated to each element of \mathcal{V}_c :

$$(f +_{\mathcal{J}} g) = [x \mapsto f(x) + g(x)];$$

- similarly the subtraction $-_{\mathcal{J}}$ is defined as:

$$(f -_{\mathcal{J}} g) = [x \mapsto f(x) - g(x) \cap \llbracket 0; +\infty \rrbracket];$$

- the synchronization $\text{SYNC}_{\mathcal{J}}(f, t)$ ensures that the abstract element f describes at least a configuration containing the threads defined by t :

$$\text{SYNC}_{\mathcal{J}}(t, f) = [x \mapsto f(x) \cup \llbracket t(x); +\infty \rrbracket];$$

- $0_{\mathcal{J}} = [x \mapsto \llbracket 0; 0 \rrbracket]$;

- $1_{\mathcal{J}}(v) = \left[\begin{cases} x \mapsto \llbracket 1; 1 \rrbracket & \text{when } x = v \\ x \mapsto \llbracket 0; 0 \rrbracket & \text{otherwise} \end{cases} \right]$.

Relational abstraction

The relational abstract domain is used to carry information between occurrence of threads or occurrence of transition labels. In contrary to the non relational abstraction, it does not constrain the computations. The synchronization primitive does not modify the abstract element.

We build a numerical abstract domain based on the ideas of KARR, proposed in [68]. An element of this domain denotes an affine subspace built on the variables \mathcal{V}_c . These elements are linked to the elements of the domain $\wp(\mathbb{N}^{\mathcal{V}_c})$ by the monotonic map $\gamma_{\mathcal{X}}$ that associates to each system its set of solutions.

COMPUTING THE UNION OF TWO AFFINE SPACES The abstract union is the most expensive and complex primitive of this domain. The principal content of [68] is the description of this operator. First of all, one can take two dual ways of representing affine subspaces. The first one, the one we choose, represents a subspace by a basis of its linear components, plus an offset vector. The second way is to represent the space as the kernel of an affine transformation from F^n to F^m for an appropriate m . Depending on the different manipulations we need and the size of the representation, the adequate method must be chosen. We take the first as the number of relationships is quite small in our context and therefore the representation is also small. Furthermore, the more union we compute, the bigger the space is and the smaller its representation.

A first step in the union computation is the definition of a canonical form for an affine subspace. As a canonical form, KARR chooses the *normalized reduced row-echelon form*.

Definition 5.1 (Normalized reduced row-echelon form) *In such representation, the matrix is in triangular form. Each first non-zero entry is equal to 1. Finally, using row operations, each first non zero entry, in our case now equal to 1, of a row is the only non zero entry of its own column.*

Once the two matrices, with their associated affine vector, denoting affine subspaces of $\mathbb{N}^{|\mathcal{V}_c|}$, are in normalized reduced row-echelon form, the union algorithm can be applied. The first idea is that affine subspaces of \mathbb{N}^n can be seen as vector subspaces of \mathbb{N}^{n+1} . Then the least affine subspace containing the two arguments of a join is computed considering the affine part of the matrices as their $|\mathcal{V}_c| + 1$ dimension.

Then the algorithm builds the resulting matrix beginning with the upper-leftmost part of the matrices. Each step considers a new column of the argument matrices. There are three cases when computing the disjunction of A and B . We denote by $A_{r,s}$ the element of A at the s -th column and on the r -th row. We consider the s -th step of computation, starting from the column 0. The three cases are depicted in Figure 5.2, page 103.

Figure 5.2 KARR's union algorithm cases.(a) $A_{rs} = B_{rs} = 1$

$$A^{(s-1)}=B^{(s-1)}= \left(\begin{array}{c|c} C^{s-1} & \begin{array}{c} \vdots \\ 0 \end{array} \\ \hline & \boxed{1} \end{array} \middle| X_0^{(s)} \right) \quad C^{(s)} \triangleq \left(\begin{array}{c|c} C^{s-1} & \begin{array}{c} 0 \\ \vdots \\ 0 \end{array} \\ \hline 0 \dots 0 & 1 \end{array} \right)$$

(b) $A_{rs} = 1; B_{rs} = 0$

$$A^{(s-1)} = \left(\begin{array}{c|c} C^{s-1} & \begin{array}{c} \vdots \\ 0 \end{array} \\ \hline & \boxed{1} \end{array} \middle| A_0^{(s)} \right) \quad B^{(s-1)} = \left(\begin{array}{c|c} C^{s-1} & \beta \\ \hline & B_0^{(s)} \end{array} \right)$$

$$C^{(s)} \triangleq (C^{s-1} | \beta)$$

(c) $A_{rs} = B_{rs} = 0$

$$A^{(s-1)} = \left(\begin{array}{c|c} C^{s-1} & \alpha \\ \hline & B_0^{(s)} \end{array} \right) \quad B^{(s-1)} = \left(\begin{array}{c|c} C^{s-1} & \beta \\ \hline & B_0^{(s)} \end{array} \right)$$

$$C^{(s)} \triangleq (C^{s-1} | \alpha) \text{ iff } \alpha = \beta$$

$A_{rs} = B_{rs} = 1$ In the first case, the next column and row to consider in both arguments of the linear disjunction are the same. They are filled with 0 except in one place: the 1 value of the first non zero entry of the new row. We adjoin the row and the column to the resulting matrix. In this case, the local iterate of the union computation preserves the exact set of solutions.

$A_{rs} = 1; B_{rs} = 0$ In the second case, one argument has the almost empty row and column (except in one entry as above) when the other has a zero entry at (r, s) . In that case, all the entry above B_{rs} are not necessary zero. Using row operations with the r -th row of A , we modify the A matrix in order to obtain the same column as in B_s . The r -th row of A is then removed. And the column, which is now the same in both matrices, adjoined to the resulting matrix.

$A_{rs} = B_{rs} = 0$ Finally the third step occurs when both columns considered are not necessary empty. If they are equal the step is immediate. But if not, we have to sacrifice a row in each matrix in order to make the two columns identical. Once the columns are the same, the rows used to perform the transformation are deleted and the new column added to the resulting matrix.

This step loses information and allows to obtain the affine subspace containing the two others.

This algorithm has an overall complexity of $O(|\mathcal{V}_c|^3)$. Expressing matrices in the canonical form costs $O(|\mathcal{V}_c|^3)$ and the disjunction algorithm costs also $O(|\mathcal{V}_c|^3)$.

OTHER PRIMITIVES The other primitives are quite straightforward:

- Computing the union of two matrices removes at least one row if they are different. As our representation considers a finite number of rows, we do not need any widening operator: we cannot compute infinitely many disjunctions without reaching a fixed point: $\nabla_{\mathcal{X}} = \cup_{\mathcal{X}}$.

- The addition $+_{\mathcal{X}}$ is defined as:

$$(O_1 + \overline{H_1}) +_{\mathcal{X}} (O_2 + \overline{H_2}) = (0_1 + {}^v 0_2) + (\overline{H_1} \cup_{\mathcal{X}} \overline{H_2})$$

where $0_1 + {}^v 0_2$ is the vector addition. In practice, we increment the right hand-side of rows containing launched threads program point.

- The subtraction $-_{\mathcal{X}}$ is similarly defined:

$$(O_1 + \overline{H_1}) -_{\mathcal{X}} (O_2 + \overline{H_2}) = (0_1 - {}^v 0_2) + (\overline{H_1} \cup_{\mathcal{X}} \overline{H_2})$$

with $0_1 - {}^v 0_2$ the vector subtraction.

- We do not constrain the synchronization in this domain. We then have:

$$\text{SYNC}_{\mathcal{X}}(t, k) = k.$$

- The element $0_{\mathcal{X}}$ is defined as the affine subspace with the following constraints:

$$0_{\mathcal{X}} = \{x = 0, \forall x \in \mathcal{V}_c\}.$$

- The primitive $1_{\mathcal{X}}(v)$ associates the value 1 to v and 0 to the other variables of \mathcal{V}_c :

$$1_{\mathcal{X}}(v) = \begin{cases} x = 1 & \text{when } x = v \\ x = 0 & \text{otherwise} \end{cases}.$$

Approximated reduced product

We now give the original reduction as proposed by FERET in [49]. The basic idea is to use, in one hand, the non relational abstraction to obtain occurrences of threads and transition labels, as well as constraining transitions if there is no sufficient enough interacting threads. But this information, associating an interval to each program point or transition label would quickly lead to unbounded occurrence numbers when computing the fixed point of the abstract collecting semantics. In the other hand, the relational abstraction can relate occurrences between each other and obtain more accurate results. The reduction of the underlying Cartesian product of the non relational domain with the relational one solves the problem. The algorithm proposed as a reduction is approximated in the sense that it does not compute the best reduction between the two elements, which could be exponential in time.

The numerical domain $\mathcal{N}_{\mathcal{V}_c}$ considered is the product $(\mathcal{I} \times \mathcal{K})$. As a product of two abstract domains, it is also a *sound* abstraction. It is partially ordered by the pairwise ordering. And it is related to element of $\wp(\mathbb{N}^{\mathcal{V}_c})$ by the concretization map $\gamma_{\mathcal{N}_{\mathcal{V}_c}}(i, k) = \gamma_{\mathcal{I}}(i) \cap \gamma_{\mathcal{K}}(k)$. The primitives $\perp_{\mathcal{N}_{\mathcal{V}_c}}, \sqcup_{\mathcal{N}_{\mathcal{V}_c}}, \nabla_{\mathcal{N}_{\mathcal{V}_c}}, +^\#, -^\#, 0^\#_{\mathcal{N}_{\mathcal{V}_c}}$ and $1^\#_{\mathcal{N}_{\mathcal{V}_c}}$ are defined pair-wisely.

The synchronization is defined as:

$$\text{SYNC}_{\mathcal{N}_{\mathcal{V}_c}}(t, (i, k)) = \rho(i', k)$$

where $i' = \text{SYNC}_{\mathcal{I}}(t, i)$.

The ρ function is the reduction operator. It is used to simplify the constraints without loosing any solution. As usual when building reduction of an abstract domain, the reduction is applied before and after each abstract transition computation.

The ρ function maps pair (i_1, k) to pair (i_2, k) leaving the KARR part untouched but relying on it to narrow intervals of i_1 .

THE INITIAL REDUCTION The reduction goal is to rely on the relational part information to narrow the effective interval associated to each program point. In that sense, it does not modify the relational abstract but only the interval part. The algorithm is in two steps. The first one aims at bounding infinite intervals. The second takes the resulting system and try to infer more precise boundaries, propagating intervals in affine equalities as in [22].

Bounding infinite intervals Let us consider an abstract element $(i, k) \in \mathcal{I} \times \mathcal{K}$ to be reduced by the algorithm. We denote by $\mathcal{V}_c^\infty \subseteq \mathcal{V}_c$ the subset of variables that are associated to an unbounded interval in i .

Let us first introduce the definition of a positive form.

Definition 5.2 (positive form) *Such form is obtained by combining rows in order to obtain, for each variable in \mathcal{V}_c^∞ , coefficients of the same sign in all constraints.*

Since we only consider positive intervals, when, in a constraint of k , all variables of \mathcal{V}_c^∞ occur with the same sign, they can be bounded using right hand-side of the affine system and the intervals associated to the variables of $\mathcal{V}_c \setminus \mathcal{V}_c^\infty$.

The positive form aims at obtaining such *valid constraints*, where all infinite variables are of the same sign. Invalid constraints are thrown away and are not used, in this approximated reduction, to improve abstract values.

The narrowing of infinite intervals is performed as follows:

1. The system is transformed by Gaussian elimination into a positive form. ($O(|\mathcal{V}_c|^3)$).
2. Once in positive form, variables of \mathcal{V}_c^∞ in all valid rows are narrowed. For a given row, when the sign of such variables is positive, we iterate through the bounded variables to compute an upper bound for these unbounded variables. Then depending on their coefficient and depending on the affine part of the constraint, a new finite upper bound is computed for each of these unbounded variables. Similarly if the sign is negative, we iterate in the considered constraints to compute the lower bound. ($O(|\mathcal{V}_c|^2)$).

Narrowing finite intervals Once the previous step is computed, we are interested in narrowing intervals. Intervals can be imprecise both because of the union computed and because of the raw infinite interval narrowing defined above.

We now focus only on constraints involving finite intervals. This second step performs an interval propagation in order to compute the approximated reduction.

1. The matrix is put in triangular form by Gaussian elimination;
2. Each affine part is replaced by its equivalent interval representation;
3. Perform as a recursive call the following steps, considering a system of constraints with an interval right hand side and a family of intervals indexed by \mathcal{V}_c .
 - a) if the system is empty, return the family of intervals as is.
 - b) else, narrow each first non zero entry of the system rows, replacing the boundaries of the associated interval using the affine part interval and the boundaries of the others variable intervals;
 - c) remove the first non zero entry, subtracting the affine interval with the one of the removed variable;
 - d) call the current algorithm on the obtained system with the narrowed interval family (each constraint has one column less);

Figure 5.3 Linear cell: an intermediate occurrence counting abstract element.

H4 : $\llbracket 0, 1 \rrbracket$	H5 : $\llbracket 1, 1 \rrbracket$	H6 : $\llbracket 0, +\infty \llbracket$	H7 : $\llbracket 0, +\infty \llbracket$	H8 : $\llbracket 1, 1 \rrbracket$
H9 : $\llbracket 1, 1 \rrbracket$	H10 : $\llbracket 1, 1 \rrbracket$	H11 : $\llbracket 1, 1 \rrbracket$	H12 : $\llbracket 0, 1 \rrbracket$	H13 : $\llbracket 0, 0 \rrbracket$
H14 : $\llbracket 1, 1 \rrbracket$	H15 : $\llbracket 1, 1 \rrbracket$	H16 : $\llbracket 0, 0 \rrbracket$		
R4 : $\llbracket 0, +\infty \llbracket$	R5 : $\llbracket 0, 0 \rrbracket$	R6 : $\llbracket 1, 1 \rrbracket$	R7 : $\llbracket 0, 0 \rrbracket$	R8 : $\llbracket 0, 0 \rrbracket$
R9 : $\llbracket 0, 0 \rrbracket$	R10 : $\llbracket 0, 0 \rrbracket$	R11 : $\llbracket 0, 0 \rrbracket$	R12 : $\llbracket 0, 0 \rrbracket$	R13 : $\llbracket 0, 0 \rrbracket$
R14 : $\llbracket 0, 0 \rrbracket$	R15 : $\llbracket 0, 0 \rrbracket$	R16 : $\llbracket 0, 0 \rrbracket$		

$$\left\{ \begin{array}{llll} H15 = 1 & H14 = 1 & H12 + R4 = 1 & H11 = 1 \\ H10 = 1 & H9 = 1 & H8 = 1 & H7 - R4 = 0 \\ H6 - R4 = -1 & H5 = 1 & H4 + R4 = 1 & R6 = 1 \\ H16, H13, R5, R7, R8, R9, R10, R11, R12, R13, R14, R15, R16 = 0 \end{array} \right.$$

In the following, to keep the presentation simple, we denote by Hx the variable $(x, T) \in \mathcal{V}_c$ and by Rx the variable $(x, F) \in \mathcal{V}_c$. H stands for threads and R for transitions.

We then focus on constraints involving only variables of \mathcal{V}_c^∞ and associated to coefficients of the same sign. The other constraints are either constraints involving only finite intervals or in undefined form, *i.e.* involving variables of \mathcal{V}_c^∞ but with coefficients of different signs.

In this example, all constraints involve either a single variable of \mathcal{V}_c^∞ or either two variables of \mathcal{V}_c^∞ but sharing the same sign.

Each non infinite part is then summarized and subtracted to the affine part (step 2). We obtain the following finite boundaries:

$$H6 : \llbracket 0, 0 \rrbracket \quad H7 : \llbracket 0, 1 \rrbracket \quad R4 : \llbracket 0, 1 \rrbracket$$

The 0 values for $H6$ appears because of the associated thread consumption in the current transition computation.

Finally we compute the second step: narrowing finite intervals by intervals propagation. We first have to identify the constraints involving only finite intervals. The previous step has inferred finite boundaries for every variables of \mathcal{V}_c^∞ . All constraints are considered.

In order to keep the presentation simple, we only focus on a subset of the constraints:

$$\left\{ \begin{array}{l} H6 - R4 = -1 \\ H7 - R4 = 0 \end{array} \right.$$

The system is already in triangular form (step 1). The affine part is replaced by its interval representation² (step 2).

$$\begin{cases} H6 - R4 = \llbracket -1, -1 \rrbracket \\ H7 - R4 = \llbracket 0, 0 \rrbracket \end{cases}$$

We try to narrow both H6 and H7 intervals with these constraints (step 3.b). The intervals associated are not yet modified. Then we remove these variables and propagate their interval (step 3.c). We obtain the following values:

$$R4 = \llbracket 1, 1 \rrbracket \cap \llbracket 0, 1 \rrbracket = \llbracket 1, 1 \rrbracket$$

The system obtained (step 3.d) gives us the value $\llbracket 1, 1 \rrbracket$ for R4 (step 3.b). This value is propagated back to the preceding system (step 3.e). We obtain the more precise information (step 3.a):

$$H7 = \llbracket 1, 1 \rrbracket$$

5.2 ENHANCING THE ABSTRACTION REDUCTION

The approximated reduction proposed above is essential to the use of the occurrence counting abstract domain. Used separately, the two underlying domains, the interval domain and the affine relationships domain gives few valuable information about occurrence properties in the analyzed term.

However the first step of the reduction algorithm may fail in reducing infinite intervals. In fact, the undefined forms obtained by the positive form transformation are thrown away and are not used anymore in the algorithm.

We first motivate this lack of precision and the necessity for a more precise narrowing of infinite intervals into finite ones. Then we describe the extension of the first step to handle more constraints.

5.2.1 Motivation

Let us continue with the example introduced above. When computing the least fixed point of the collecting semantics of the term 5.1, we obtain this other intermediate abstract element, given in Figure 5.4

In that case, the unbounded intervals are $\mathcal{V}_c^\infty = \{H7, H8, H10, H16, R5, R6, R15\}$.

H16, H10, H8, H7, R15 appear in only one constraint each and are therefore already in positive form.

² The intervals used in this part of the algorithm could denote negative values.

Figure 5.4 Linear cell: another intermediate occurrence counting abstract element.

$$\begin{array}{llllll}
 \text{H4:} & \llbracket 0, 1 \rrbracket & \text{H5:} & \llbracket 1, 1 \rrbracket & \text{H6:} & \llbracket 0, 1 \rrbracket & \text{H7:} & \llbracket 0, +\infty[& \text{H8:} & \llbracket 0, +\infty[\\
 \text{H9:} & \llbracket 0, 1 \rrbracket & \text{H10:} & \llbracket 0, +\infty[& \text{H11:} & \llbracket 0, 1 \rrbracket & \text{H12:} & \llbracket 0, 1 \rrbracket & \text{H13:} & \llbracket 0, 1 \rrbracket \\
 \text{H14:} & \llbracket 0, 1 \rrbracket & \text{H15:} & \llbracket 1, 1 \rrbracket & \text{H16:} & \llbracket 0, +\infty[& & & & \\
 \text{R4:} & \llbracket 0, 1 \rrbracket & \text{R5:} & \llbracket 0, +\infty[& \text{R6:} & \llbracket 0, +\infty[& \text{R7:} & \llbracket 0, 0 \rrbracket & \text{R8:} & \llbracket 0, 0 \rrbracket \\
 \text{R9:} & \llbracket 0, 0 \rrbracket & \text{R10:} & \llbracket 0, 0 \rrbracket & \text{R11:} & \llbracket 0, 0 \rrbracket & \text{R12:} & \llbracket 0, 0 \rrbracket & \text{R13:} & \llbracket 0, 0 \rrbracket \\
 \text{R14:} & \llbracket 0, 1 \rrbracket & \text{R15:} & \llbracket 0, +\infty[& \text{R16:} & \llbracket 0, 0 \rrbracket & & & &
 \end{array}$$

$$\left\{ \begin{array}{ll}
 \text{H16} - \text{R14} = 0 & \text{H15} = 1 \\
 \text{H14} + \text{R14} = 1 & \text{H13} + \text{H11} = 1 \\
 \text{H12} + \text{H10} + \text{R4} + \text{R5} - \text{R6} = 1 & \text{H9} + \text{R5} - \text{R6} = 0 \\
 \text{H8} - \text{R6} + \text{R14} + \text{R15} = 0 & \text{H7} - \text{R4} - \text{R5} = 0 \\
 \text{H6} - \text{R4} - \text{R5} + \text{R6} = 0 & \text{H5} = 1 \\
 \text{H4} + \text{R4} = 1 & \\
 \text{R7, R8, R9, R10, R11, R12, R13, R16} = 0 &
 \end{array} \right.$$

Computing a positive form for the matrix gives:

$$\left\{ \begin{array}{l}
 \text{H16} - \text{R14} = 0 \\
 \text{H12} + \text{H10} + \text{R4} + \text{R5} - \text{R6} = 1 \\
 \text{H9} + \text{R5} - \text{R6} = 0 \\
 \text{H8} - \text{R6} + \text{R14} + \text{R15} = 0 \\
 \text{H9} + \text{H7} - \text{R4} - \text{R6} = 0 \\
 \text{H9} + \text{H6} - \text{R4} = 0
 \end{array} \right.$$

When computing such positive form, the combinations leading to only positive coefficients for R5 consume the constraint with positive coefficients for R6. We are not able to obtain a positive form with positive coefficients for both R5 and R6.

Let us now find out which constraint can be used to bound infinite intervals. Only the first one contains variables of \mathcal{V}_c^∞ with the same sign. In that example, four of the five constraints of interest are undefined forms with respect to the positive form definition.

However these affine constraints could be used together and allow us to produce new ones satisfying both the positive form and the constraint of having only variables of \mathcal{V}_c^∞ associated to coefficients of the same sign.

The new section addresses such improvement.

5.2.2 The reduction revisited

The reduction algorithm is modified in order to deal with more constraints. Particularly with some undefined forms generated by the positive form transformation.

We only modify the first step of the algorithm which try to narrow infinite intervals into finite ones. As in the original presentation, we target an approximated reduction which does not capture all possible reduction but handle as much as possible while keeping a reasonable complexity. The enhancement we propose does not change the overall complexity of the reduction while producing more *good* constraints for the reduction of infinite intervals.

The first step is reformulated as:

Bounding infinite intervals revisited

- We repeat until reaching an empty system:
 1. The system is transformed by Gaussian elimination into a positive form. Such form is obtained by combining rows in order to obtain, for each variable in \mathcal{V}_c^∞ , coefficients of the same sign in all constraint. ($O(|\mathcal{V}_c|^3)$).
 2. Solve valid rows as previously, *i.e.* involving only variables of \mathcal{V}_c^∞ sharing the same coefficients sign. And remove solved rows from the system, keeping only undefined constraints. ($O(|\mathcal{V}_c|^2)$).
 3. Variables of \mathcal{V}_c^∞ are identified using the current interval abstract element. The columns associated to these variables are put at the beginning of the matrix. ($O(|\mathcal{V}_c^\infty|)$).
 4. Put the system in canonical form ³. ($O(|\mathcal{V}_c|^3)$).
 5. Eliminate the first variable of the last constraint, by Gaussian combination, removing this last constraint from the resulting matrix.

The last step that eliminates the last row allows to solve the undefined forms. The last row involves at least two variables in \mathcal{V}_c^∞ with coefficients of different signs. Otherwise the constraint must be solved before and therefore removed. The elimination of this first variable of the last constraint in other constraints produces new constraints with possibly coefficients of the same sign for variables remaining in \mathcal{V}_c^∞ .

The first step of the reduction as presented here has a worst case complexity of $O(m \times |\mathcal{V}_c|^3)$ where m is the number of constraints of the initial system. However the algorithm can be reformulated in order to be still in $O(|\mathcal{V}_c|^3)$. The unbounded interval variables are initially identified, the column exchanged, the canonical form computed as well as the initial positive form. Then the same

³ The canonical form is the normalized reduced row-echelon form as presented in Section 5.1.1.

schemata is applied, the positive form as well as the canonical matrix being kept and locally modified when solving rows or removing the last row.

5.2.3 The example reconsidered

We now apply our modified flavor of the algorithm first step on our previous example. We recall that initially $\mathcal{V}_c^\infty = \{H7, H8, H10, H16, R5, R6, R15\}$.

We first solve valid constraints. Let us now consider the remaining system. The set of unbounded variables is now $\mathcal{V}_c^\infty = \{H7, H8, H10, R5, R6, R15\}$. We permute columns in order to get variables of \mathcal{V}_c^∞ first:

$$\left\{ \begin{array}{l} H10 + R5 - R6 + H12 + R4 = 1 \\ R5 - R6 + H9 = 0 \\ H8 - R6 + R15 + R14 = 0 \\ H7 - R5 - R4 = 0 \\ -R5 + R6 + H6 - R4 = 0 \end{array} \right.$$

Then the last one is removed from the system and eliminated from the other constraints:

$$\left\{ \begin{array}{l} H10 + H12 + H6 = 1 \\ H9 + H6 - R4 = 0 \\ H8 - R6 + R15 + R14 = 0 \\ H7 - R6 - H6 = 0 \end{array} \right.$$

Then we transform the resulting system into positive form and try to bound valid constraints. The system is already in positive form and the first constraint satisfies the same signs predicate.

H10 is now bounded by $\llbracket 0, 2 \rrbracket$.

The algorithm continues to run on the non empty system but does not gives more boundary information.

5.3 CONSIDERING COMPUTED TRANSITIONS

We propose in this section a modification of the first occurrence abstraction $\mathbb{N}^{\mathcal{V}_c}$. We first motivate this modification. Then we state the new abstract domain and associated semantics. Finally we illustrate its use on our example.

5.3.1 Motivation

The main occurrence counting abstraction presented here allows to represent with care the effect of computing transitions on occurrences of threads in reachable configurations.

Figure 5.5 Linear cell: an intermediate occurrence counting abstract element computing a spurious transition.

H4 :	$\llbracket 0, 1 \rrbracket$	H5 :	$\llbracket 1, 1 \rrbracket$	H6 :	$\llbracket 0, 0 \rrbracket$	H7 :	$\llbracket 0, +\infty \llbracket$	H8 :	$\llbracket 0, +\infty \llbracket$
H9 :	$\llbracket 0, 0 \rrbracket$	H10 :	$\llbracket 0, 1 \rrbracket$	H11 :	$\llbracket 0, 1 \rrbracket$	H12 :	$\llbracket 1, 1 \rrbracket$	H13 :	$\llbracket 0, 1 \rrbracket$
H14 :	$\llbracket 0, 1 \rrbracket$	H15 :	$\llbracket 1, 1 \rrbracket$	H16 :	$\llbracket 0, 1 \rrbracket$				
R4 :	$\llbracket 0, 0 \rrbracket$	R5 :	$\llbracket 0, +\infty \llbracket$	R6 :	$\llbracket 0, +\infty \llbracket$	R7 :	$\llbracket 0, 0 \rrbracket$	R8 :	$\llbracket 0, 0 \rrbracket$
R9 :	$\llbracket 0, 0 \rrbracket$	R10 :	$\llbracket 0, 0 \rrbracket$	R11 :	$\llbracket 0, 0 \rrbracket$	R12 :	$\llbracket 0, 0 \rrbracket$	R13 :	$\llbracket 0, 0 \rrbracket$
R14 :	$\llbracket 0, 1 \rrbracket$	R15 :	$\llbracket 0, +\infty \llbracket$	R16 :	$\llbracket 0, 0 \rrbracket$				

Let us focus on the synchronization step that allows or not a transition to be computed. The abstract semantics mimics the concrete transition: the transition is computable if the abstract element contains interacting threads.

However this abstract semantics could lead to spurious transition that can easily be identified as unfeasible when looking at the term. We recall that a transition occurs if all abstract elements in Cartesian product allow it. Let us consider the case where the values (or their abstraction) associated to some threads environments allow the computation, and when these threads were produced at least once during the preceding abstract transitions. The union computation may have lost causality expressed by the relational part of the occurrence counting abstraction.

In the example above, the transition between the thread at program point 4 (the initial linear cell in its put state) and the threads at program point 10, denoting put messages generated after the get message receiving must never occurs. Trivially the generation of any thread at program point 10 necessitates the consumption of the initial thread at program point 4.

Using the preceding occurrence counting abstraction, we obtain the following abstract element for intervals, given in Figure 5.5. In particular,

$$H4 : \llbracket 0, 1 \rrbracket \quad H10 : \llbracket 0, 1 \rrbracket$$

After both the synchronization and the reduction computation, the only modified intervals are

$$H4 : \llbracket 1, 1 \rrbracket \quad H10 : \llbracket 1, 1 \rrbracket$$

No interval is associated to the $\perp_{\mathcal{S}}$ value: the abstract transition can occur where it should not. The abstraction, even reduced, is imprecise and allows a spurious communication.

5.3.2 *Abstract domain*

We propose to enrich the abstract element with direct dependencies between transitions and launched threads. We only focus on the first launching of threads and do not carry information about the recursive launching of threads.

The new abstraction of occurrence counting is built upon the previous abstraction. We first build an abstract domain carrying abstract dependencies. This domain is parametrized by another abstract domain. It constrains the synchronization of the underlying domain.

The abstract semantics is modified in order to:

- consider dependencies when computing synchronization;
- update dependencies when launching threads.

A generic abstract domain

We introduce the set $\mathcal{Dep} = \wp(\mathcal{L}_a \times \mathcal{L}_p)$ of dependencies among threads program points. This domain is fitted with a pre-order relation $\sqsubseteq_{\mathcal{Dep}}$ defined pairwise using usual set inclusion. Similarly the abstract union $\sqcup_{\mathcal{Dep}}$ is the union of sets and the empty set denotes the bottom element $\perp_{\mathcal{Dep}}$. A widening operator is not necessary since the set \mathcal{Dep} only admits finite ascending sequences. The initial element $C_0^{\mathcal{Dep}}$ is defined as the empty set.

We relate elements of \mathcal{Dep} to set of concrete configurations using the monotonic map $\gamma_{\mathcal{Dep}}$:

$$\{\epsilon, C_0\} \cup \left\{ (u, C) \left| \begin{array}{l} \forall u', u'. \lambda \text{ prefixes of } u \text{ such that} \\ (u', C') \xrightarrow{\lambda} (u'.\lambda, C' \cup \text{new} \setminus \text{removed}) \\ \text{and } \lambda \text{ is a static transition} \\ \implies \forall (p, \text{id}, E) \in \text{new}, (p^a, p) \in \text{dep}^\# \\ \text{where } p^a \text{ the actor program point in } \lambda \end{array} \right. \right\} \subseteq \gamma_{\mathcal{Dep}}(\text{dep}^\#)$$

The pair $(n, n') \in \text{dep}^\#$ denotes that the transition labeled by program point n launches the thread associated to the program point n' .

Let us consider an occurrence counting abstract domain $(N, \sqsubseteq_N, \sqcup_N, \perp_N, \gamma, C_0^N, \rightarrow_N, \nabla_N)$. The main abstract domain pre-order \sqsubseteq , union operator \sqcup , bottom element \perp , and initial abstract $C_0^\#$ are then defined pairwise. The widening ∇ is defined as follows:

$$(d, o) \nabla (d', o') = (d \sqcup_{\mathcal{Dep}} d', o \nabla_N o')$$

The concretization $\gamma(d, o)$ is defined as in Cartesian product as $\gamma_{\mathcal{Dep}}(d) \cap \gamma_N(o)$.

We define our generic main occurrence counting abstraction as

$$(\mathcal{Dep} \times N, \sqsubseteq, \sqcup, \perp, \gamma, C_0^\#, \rightarrow_\#, \nabla)$$

Abstract operational primitives

In order to define the abstract transition $\rightarrow_{\#}$, let us first introduce abstract primitives.

ABSTRACT SYNCHRONIZATION The synchronization step ensures that not only interacting threads are present in the over-approximation of reachable configurations but also that the approximation has computed the necessary transitions. We first introduce the equivalence class computation based on the dependency relations of the abstract element. Then the synchronization primitive is defined, relying on the synchronization of the underlying numerical abstract domain. It modifies the constraints using the associated equivalence class.

Definition 5.3 (Dependencies transitive closure) We define the dependencies set $\text{dep}(p, d^{\#}) \subseteq \mathcal{L}_p$ for a given program point $p \in \mathcal{L}_p$ and an abstract dependency $d^{\#} \in \text{Dep}$ as the transitive closure of the following relation \rightsquigarrow :

$$p_1 \rightsquigarrow p_2 \text{ iff } (p_1, p_2) \in d^{\#}$$

The dependencies closure allows to constrain the computed transitions needed for the existence of the considered interacting threads

Definition 5.4 (Synchronization) The abstract synchronization relies on the dependencies in order to consider computed transitions when ensuring that the numerical abstraction contains sufficient threads. Let (d, o) be an abstract element of $\text{Dep} \times \mathbb{N}$ and $t \in \mathcal{L}_p$ be a set of program points. The primitive SYNC is defined as follows:

We denote by t' the family of natural numbers indexed by \mathcal{V}_c such that t'_v is the equal to one if $v = \begin{cases} (p, T) & p \in t; \\ (p, F) & \exists p' \in t, p \rightsquigarrow^+ p'. \end{cases}$

$$\text{SYNC}(t, (d, o)) \triangleq (d, \text{SYNC}_N(t', o))$$

where SYNC_N denotes the synchronization primitive of the occurrence counting domain.

LAUNCHING THREADS The abstract transition is computable, *i.e.* the other domains allow it and the current underlying numerical abstraction with the modified synchronization constraints enforced is not equal to \perp . Then we have to reflect the thread launching on the current abstract element. We only focus on the first launching of threads.

In the CAP case, continuation threads can be launched either by the syntactic actor defining them or by the use of its behavior branch with a dynamic actor. Therefore, we only store information concerning transition that do not involve replication threads.

Definition 5.5 (Thread launching) *The abstract launch builds a new dependence between the transition label and the launched thread when involving only consumed threads. Let $(p^k)_k$ be a tuple of interacting program points, $(s^k)_k$ the partial interaction type associated to each interacting thread and $(\text{cont}^k)_k$ the continuation of each associated partial interaction.*

We denote by $\text{pi_type}(s) \in \{\text{computation}, \text{replication}\}$ the type of the partial interaction type s . We define the `launch_dep` primitive as:

$$\text{launch_dep}((p^k), (s^k), (\text{cont}^k)) \triangleq \begin{cases} \{p^1\} \times \text{cont_pp}^1 & \text{when } \text{pi_type}(s^1) \neq \text{replication} \\ \emptyset & \text{otherwise} \end{cases}$$

where cont_pp^1 denotes the set of program points associated to static threads of cont^1 .

Abstract operational semantics

The initial state is defined as the empty set of dependencies and the initial element for the underlying numerical abstraction.

We now detail the computation of $\frac{(p^k)_k}{\#}$. The abstract transition mimics the concrete transition. The synchronization relies on the synchronization of the underlying domain. Launching continuations updates both the dependencies and the numerical element.

The operational semantics is given in Figure 5.6.

Theorem 5.6 *The abstraction defined by $(\text{Dep} \times \mathbb{N}, \sqsubseteq, \sqcup, \perp, \gamma, C_0^\#, \rightarrow_\#, \nabla)$ is a sound abstraction with respect to Definition 3.3.*

Proof 5.7 *The proof can be found in Annexe A.3.*

5.3.3 The example reconsidered

Let us consider again the above example with the modified occurrence counting. The abstract element considered also contains the following dependencies:

$$R14 : \{H16\} \quad R6 : \{H11, H10, H9, H8\} \quad R4 : \{H7, H6\}$$

The synchronization enforces the presence not only of H4 and H10 but also ensures the past computation of transitions labeled by R4 and R6.

After computing both the synchronization and the reduction with the relational part, we obtain:

Figure 5.6 Abstract operational primitive for the occurrence counting domain with transitions.

Let $(d, o)^\# \in \mathcal{D}_{ep} \times \mathbb{N}$ be an abstract configuration, let \mathcal{R} be a formal rules, let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program points label and $(pi^k)_{1 \leq k \leq n} = (s^k, (par^k), (bd^k), cont^k)$ be a tuple of partial interactions. We define by t the set of interacting program points $\{(p^k)_{1 \leq k \leq n}\}$.

When

1. $\forall k \in \llbracket 1; n \rrbracket, pi^k \in \text{interaction}(p^k)$;
2. and $\text{SYNC}(t, (d, o)^\#) \neq \perp$.

Then

$$(d, o)^\# \xrightarrow{\mathcal{R}, (p^k), (pi)^k} \#(d \cup_{\mathcal{D}_{ep}} \text{new_dep}, o')^\#$$

Where

1. $\text{new_dep} = \text{launch_dep}((p^k), (s^k), (cont^k))$;
 2. o' is defined as $o \xrightarrow{\mathcal{R}, (p^k), (pi)^k} \#o'$.
-

H4 : $\perp_{\mathcal{J}}$	H5 : $\llbracket 1, 1 \rrbracket$	H6 : $\perp_{\mathcal{J}}$	H7 : $\llbracket 0, +\infty \llbracket$	H8 : $\llbracket 0, +\infty \llbracket$
H9 : $\perp_{\mathcal{J}}$	H10 : $\llbracket 1, 1 \rrbracket$	H11 : $\llbracket 0, 1 \rrbracket$	H12 : $\perp_{\mathcal{J}}$	H13 : $\llbracket 0, 1 \rrbracket$
H14 : $\llbracket 0, 1 \rrbracket$	H15 : $\llbracket 1, 1 \rrbracket$	H16 : $\llbracket 0, 1 \rrbracket$		
R4 : $\perp_{\mathcal{J}}$	R5 : $\llbracket 0, +\infty \llbracket$	R6 : $\llbracket 0, +\infty \llbracket$	R7 : $\llbracket 0, 0 \rrbracket$	R8 : $\llbracket 0, 0 \rrbracket$
R9 : $\llbracket 0, 0 \rrbracket$	R10 : $\llbracket 0, 0 \rrbracket$	R11 : $\llbracket 0, 0 \rrbracket$	R12 : $\llbracket 0, 0 \rrbracket$	R13 : $\llbracket 0, 0 \rrbracket$
R14 : $\llbracket 0, 1 \rrbracket$	R15 : $\llbracket 0, +\infty \llbracket$	R16 : $\llbracket 0, 0 \rrbracket$		

The $\perp_{\mathcal{J}}$ appears in the resulting abstract element: the transition is not computable. In the considered example, both transitions (4, 10) and (5, 9, 12) are never computed. The resulting element is therefore more precise, involving less spurious transitions when computed.

5.4 REDUCING TRANSITION OCCURRENCES BETWEEN PARTITION UNITS

Finally the last enhancement concerning occurrence counting is specific to the use of these domains under the partitioned abstract domain presented in Chapter 4.

The partitioned abstract domain presented in the previous chapter associates to each name binder the properties specific to threads bound to this address,

merging recursive instances of addresses in the same partition unit abstract element.

Using occurrence counting domains under the partitioned abstraction produces new constraints, specific to a given binder, that could not be observed globally. In fact the union computation of abstract element weaken the properties and could miss some key information. For example the numerical properties concerning a thread $x \triangleright s$ where x can be bound to different binders are merged when computing the disjunction. Using occurrence counting in a partitioned view could separate the numerical properties specific to each binder and allow to prove the properties of interest.

In this section, rather than introducing a new domain, we propose a global organization of abstract domains in order to obtain both precise information locally for a given binder and globally for all threads and transition labels.

5.4.1 *Combining abstract domains*

The main abstract domain proposal is a product of an occurrence counting abstraction with a partitioned abstract domain.

The occurrence counting domain carries numerical properties globally for all threads and transition labels.

The partitioned domain contains both the control flow domain and the domain which elements are associated to each partition units. We recall that, in this context, the control flow abstract domain drives the abstract transitions and allows to over-approximate partition units. The underlying partitioned domain is here the occurrence counting domain. Each address binder is associated to the numerical properties of its threads occurrences.

5.4.2 *Reduction*

The abstract semantics of the partitioned domain splits the threads to be launched into the possible units they target. Each unit is then updated considering the possibly empty set of continuation threads. In this view, each unit is updated considering the thread launched and the computed transition.

Then this semantics updates units with the incremented transition label even if they do not contain new launched threads. All transition label intervals are therefore equal among local elements and the global one.

We propose the following reduction in order to rely on both the global and local properties computed.

Definition 5.8 (Two-ways Reduction)

- We reduce global intervals associated to transition labels with their associate local intervals:

$$\forall i_{\text{loc}} \in \text{IntervalsUnits}, \forall x \in \mathcal{V}_c, i_{\text{glob}}(x, \mathbb{F}) \triangleq i_{\text{glob}}(x, \mathbb{F}) \sqcap_{\mathcal{I}} i_{\text{loc}}(x, \mathbb{F})$$

where IntervalsUnits denotes the set of intervals associated to the different partition units and i_{glob} denotes the intervals of the global occurrence counting abstract element.

- We reduce back intervals of numerical transitions in partition units with the global information obtained:

$$\forall i_{\text{loc}} \in \text{IntervalsUnits}, \forall x \in \mathcal{V}_c, i_{\text{loc}}(x, \mathbb{F}) \triangleq i_{\text{glob}}(x, \mathbb{F}) \sqcap_{\mathcal{I}} i_{\text{loc}}(x, \mathbb{F}).$$

5.5 SUMMARY

In this chapter, we proposed three enhancements concerning the use of occurrence counting abstraction within this framework of abstract interpretation of mobile system. All these enhancements are not CAP specific and can be applied to any process calculus encoded in the framework.

5.5.1 Contributions

The contributions are the following:

REDUCTION BETWEEN AFFINE RELATIONSHIPS AND INTERVALS We extend the approximated reduction between KARR's abstract domain of affine relationship and the intervals to handle more cases. In particular we modify the first step that reduces infinite intervals into finite ones. The modified algorithm relies on the original version and combines undefined forms by Gaussian elimination to produce good ones. These new constraints can then be used using the original algorithm and more intervals narrowed.

One can notice that the refined algorithm can also be easily applied to narrow intervals of \mathbb{Z} . In that case, the infinite reduction is computed in two times, dealing with the right hand-side boundary first then the left hand-side. The narrowing of finite intervals is keep untouched.

CONSIDERING COMPUTED TRANSITIONS We modified the global occurrence counting in order to consider computed transition and avoid spurious transitions. This new occurrence counting domain forbids the transition between threads when one depends on the other thread consumption to be created. The dependencies among recursive threads were already expressed using the relational abstraction but these direct relations between the initial thread and the thread produced the first time were not considered in the previous domain.

This enhancement avoids a lot of spurious abstract transitions that would then weaken the result abstract properties.

REDUCTION WHEN USED IN THE PARTITIONED ABSTRACT DOMAIN Based on the partitioned abstract domain presented previously, we exploit this partitioning of properties by name binder. Using the occurrence counting domain under the partitioning produces constraints that could not be obtain globally otherwise. However the fixed point computation, with its widening, locally produces unbounded intervals that could not always be narrowed into finite ones by the local relational properties. The reduction we introduced globally constrains the intervals counting transition label occurrences. Then the reduced intervals constrain the other thread intervals using the local relational properties.

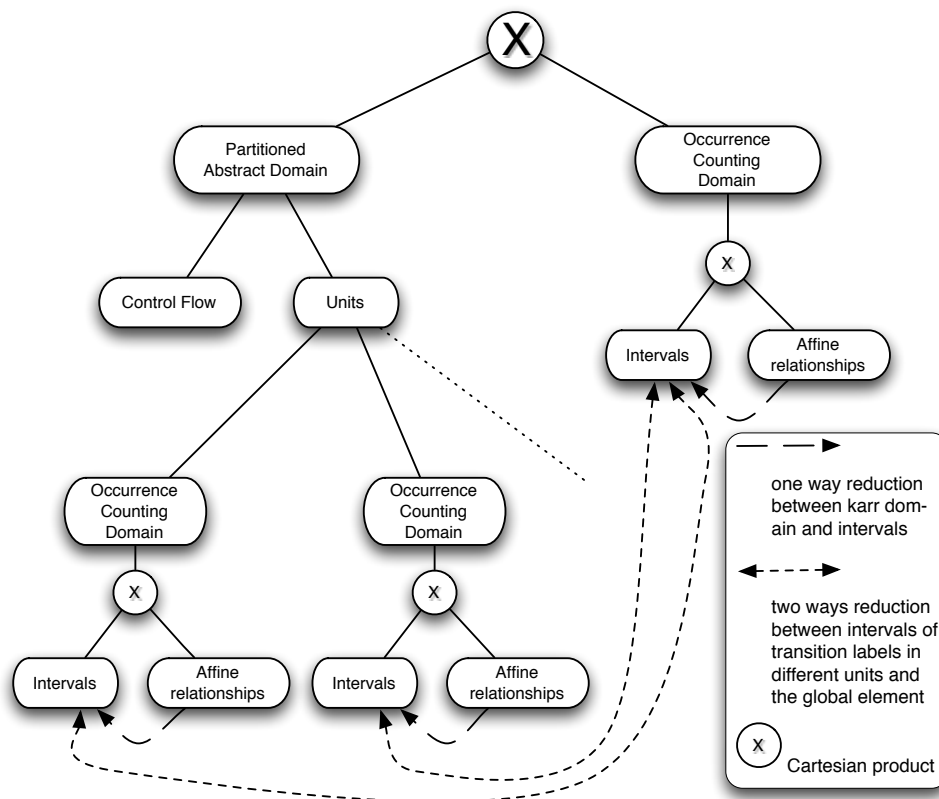
5.5.2 *General overview*

All the enhancements proposed do not change the complexity class of basic operators of the occurrence counting abstractions. All operators, including reduction computations, stay in the worst-case time complexity of $O(|\mathcal{V}_c|^3)$.

The Figure 5.7 gives an overview of the abstraction used to over-approximate occurrence counting properties.

The use of the occurrence counting abstraction gives great information on the numerical properties of reachable configurations. In general, other verification techniques such as type systems or model checking do not handle well such properties, either because they are not precise enough to model them, or either because they do not scale well. The proposed domains as well as the enhancements introduced give a very parametric way to use existing numerical domains in such framework. Both non relational and relational numerical domains can then be used as basis domains. The reduction algorithms as well as the associated abstract operational semantics presented here guaranty to obtain precise results. One can then observe boundedness properties of the system or even mutual exclusion between threads.

Figure 5.7 Overview of the global occurrence counting abstraction.



This chapter addresses the analysis of the linearity property for CAP terms. Linear terms consider addresses as resources and do not allow multiple actors associated to the same address at a given time.

CAP syntax and semantics would allow such terms to be written but we consider them as invalid. The objective of this chapter is to propose an analysis that could guarantee a term and its evolutions to be linear.

This property analysis is essential in CAP since we always want to consider linear systems. A CAP term with at least two actors bound on the same address does not model anything meaningful and should not be considered. Furthermore the analysis of any other property specific to CAP relies on this linearity assumption.

Among the difficulties of such an analysis, the behavior passing mechanism makes the analyses complex as it becomes difficult to precisely track the dynamic topology of communications in such a high-order paradigm.

The first section states the problematics, gives the property definition and illustrates it. The Section 6.2 proposes an abstract domain to track the use of each variable value in order to ensure linearity. An example is given and the detail of resulting abstract elements computation is presented in Section 6.3. Finally, the last two Sections 6.4 and 6.5 compare our analysis to related approaches and conclude.

An initial version of this work was presented in [54], targeting the comparison of type system-based and abstract interpretation-based linearity checking.

6.1 PROBLEMATICS

Linearity in concurrent calculi and more generally in programming languages is a very important concern. Depending on the context, linear systems can be optimized or implemented in a memory-efficient way.

Linearity in CAP is essential as it allows to consider addresses as exclusive resources like IP addresses in networks. Its main purpose is to ensure that at any time only one behavior is reading messages from an actor mailbox. Such an analysis is not trivial in CAP since an address may be dynamically associated to many behaviors in its lifetime.

In this section we first state the definition of linear terms and then propose a first approach to the linearity checking, considering already existing information in the abstract elements of previously presented analyses.

6.1.1 *Definition*

Depending on the considered term, linearity can be observed more or less easily. In the Example 2.7 page 27, we can easily observe that when the term cannot be reduced anymore, there is two actors bound on address a . In contrary, the configuration given page 29 in Example 2.9 is more complex and necessitates a precise analysis to ensure that any reachable configuration satisfies the linearity property for all existing actors.

LINEAR TERMS

Definition 6.1 (Linear terms) Let $C \in \mathcal{C}$ be a CAP configuration, we denote by $\text{LinearFailure}(C, a)$ the property of the term to be non linear for address a :

$$\text{LinearFailure}(C, a) \triangleq C \equiv \nu a, a \triangleright [\dots] \parallel a \triangleright [\dots] \parallel \dots$$

Then we define the linearity property: a term is said to be linear if every reachable state does not break the linearity for each possible actor address.

$$\text{Linear}(C) \triangleq \forall C' \in \mathcal{C}, \forall a \in \mathcal{N} : C \rightarrow^* C' \implies \neg \text{LinearFailure}(C', a)$$

6.1.2 *Examples*

Example 7 The example trace given in Figure 6.1 illustrates a non linear system.

Initially, in this system, two actors are bound to addresses a and b and two messages labeled m with the same argument c are sent to the address b . All the maximal traces of this example lead to a non linear configuration.

The non linearity arises because the actor on a freely installs actors on the argument value of received messages, and replicates. The two initial messages are sent to the actor on b which forwards them to the actor on a . Once the second one is transmitted, the configuration has two actors bound to the same address c .

6.1.3 *A first attempt of linearity checking in our framework*

An immediate solution is to rely on the existing information computed by the abstract interpretation framework in both control flow approximation, in particular the shape abstract domain, and occurrence counting abstraction.

Relying on control flow and occurrence counting properties

We first over-approximate the set of actor thread program points that can be associated to the same address. The result of the shape analysis presented in Section 3.3.2 gives us this information.

Figure 6.1 Non linearity example trace.

$$\begin{array}{l}
 \nu a^\alpha, \nu b^\beta, \nu c^\gamma, \quad a \triangleright^1 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel b \triangleright^5 [m^6(x) = \zeta(e, s)(e \triangleright^7 s \parallel a \triangleleft^8 n(x))] \\
 \quad \parallel b \triangleleft^9 m(c) \parallel b \triangleleft^{10} m(c) \\
 \\
 \xrightarrow{(5,9)} \quad \nu a^\alpha, \nu b^\beta, \nu c^\gamma, \quad a \triangleright^1 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel b \triangleright^7 [m^6(x) = \zeta(e, s)(e \triangleright^7 s \parallel a \triangleleft^8 n(x))] \\
 \quad \parallel a \triangleleft^8 n(c) \parallel b \triangleleft^{10} m(c) \\
 \\
 \xrightarrow{(1,8)} \quad \nu a^\alpha, \nu b^\beta, \nu c^\gamma, \quad a \triangleright^3 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel b \triangleright^7 [m^6(x) = \zeta(e, s)(e \triangleright^7 s \parallel a \triangleleft^8 n(x))] \\
 \quad \parallel b \triangleleft^{10} m(c) \\
 \quad \parallel c \triangleright^4 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \\
 \xrightarrow{(7,10)} \quad \nu a^\alpha, \nu b^\beta, \nu c^\gamma, \quad a \triangleright^3 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel b \triangleright^7 [m^6(x) = \zeta(e, s)(e \triangleright^7 s \parallel a \triangleleft^8 n(x))] \\
 \quad \parallel a \triangleleft^8 n(c) \\
 \quad \parallel c \triangleright^4 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \\
 \xrightarrow{(3,8)} \quad \nu a^\alpha, \nu b^\beta, \nu c^\gamma, \quad a \triangleright^3 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel b \triangleright^7 [m^6(x) = \zeta(e, s)(e \triangleright^7 s \parallel a \triangleleft^8 n(x))] \\
 \quad \parallel c \triangleright^4 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)] \\
 \quad \parallel c \triangleright^4 [n^2(x) = \zeta(e, s)(e \triangleright^3 s \parallel x \triangleright^4 s)]
 \end{array}$$

In order to compute these sets of actors that potentially share a same address, we define the equivalence relation \sim as the strongest equivalence relation that relates actors on the same address. We first define a relation $\sim' \in \mathcal{L}_p^2$ between two program points of a term considering the abstract element resulting of the over-approximation of the term collecting semantics.

Definition 6.2 (Actor sharing their address) $\forall p, q \in \mathcal{L}_p, p \sim' q$ iff

- *there exists $(s^p, (\text{parameters}_i^p), (\text{bounded}_i^p), \text{continuation}^p) \in \text{interaction}(p)$ such that $s^p \in \{\text{static_actor}_n, \text{dynamic_actor}\}$;*
- *similarly, there exists $(s^q, (\text{parameters}_i^q), (\text{bounded}_i^q), \text{continuation}^q) \in \text{interaction}(q)$ such that $s^q \in \{\text{static_actor}_n, \text{dynamic_actor}\}$;*
- $\text{shape}(p, \text{parameters}_1^p) \sqcap_{\mathcal{L}_p}^{\text{Reg}} \text{shape}(q, \text{parameters}_1^q) \neq \perp_{\mathcal{L}_p}^{\text{Reg}}$;

where $\text{shape}(p, x)$ denotes the element of $\text{Reg}_{\mathcal{L}_p}$ associated to the variable x in the shape approximation of the program point p .

The equivalence relation \sim can be defined as the strongest equivalence relation that is compatible with \sim' .

Definition 6.3 (Address sharing equivalence relation) We define $\sim \in \mathcal{L}_p^2$ as the reflexive, symmetric and transitive closure of the relation $\sim' \in \mathcal{L}_p^2$. Thus the relation is an equivalence relation. For any program point $p \in \mathcal{L}_p$, we denote by $[p]_{\sim}$ its equivalence class (i.e. $[p]_{\sim} = \{q \in \mathcal{L}_p \mid p \sim q\}$). We denote by C_{\sim} the set of all equivalence classes.

Each equivalence class of C_{\sim} is then checked to ensure linearity. Actors of the same equivalence class must be present at most once and must be in mutual exclusion.

Definition 6.4 (Linearity checking using occurrence properties) *The term is linear iff*

- *each actor is present at most once in each reachable configuration:*

$$\text{Intervals}(p, T) \sqsubseteq_{\mathcal{I}} \llbracket 0, 1 \rrbracket$$

where $\text{Intervals} : \mathcal{V}_c \rightarrow \mathbb{N}^2$ is the non relational part of the resulting occurrence counting abstract element that maps threads program points, i.e. (p, T) , and transition labels, i.e. (p, F) , to intervals;

- *all actors in the same equivalence class $[p]_{\sim}$ are in mutual exclusion. This property can be observed in the Karr resulting abstract element of the occurrence counting abstraction.*

We look for constraints expressing mutual exclusion between actors, such as:

$$\sum_{p_i \in [p]_{\sim}} p_i = 1$$

Other constraints involving other positive occurrences of threads or transition can also denote mutual exclusion:

$$X + \sum_{p_i \in [p]_{\sim}} p_i = 1$$

where X only contains positive multiplicities.

Relying on address partitioning

The proposed approach can be enhanced using the partitioned abstract domain. The set of actors associated to a given name binder can be here easily computed. But it can be less precise than the equivalence classes obtained using the shape marker analysis.

However the occurrence counting abstract domain used under the partitioned abstract domain gives more relevant properties about occurrences of actor threads for a given name binder.

The linearity checking is then modified:

1. Using control flow abstract element, we compute equivalence classes.
2. Each equivalence class is associated to the set of name binders it can be associated to.
3. Then the occurrence constraints (one thread at most, and mutual exclusion) are checked for each equivalence class and each possible binder, considering the occurrence counting abstract element of the partition unit associated to the binder.

6.2 ABSTRACTING LINEARITY

The previous analysis do not give often accurate results. We now present an abstract domain to be used, in our abstract interpretation-based framework for CAP, in order to check that a term satisfies the linearity requirement.

6.2.1 *Intuition*

This domain allows us to express linearity in the sense of the type system defined in [26] and presented at the end of the chapter.

Our semantics is defined by an abstract transition relation. In fact, the current relation has two ways of data-flow, expressing both the launching of new

threads and the updating of interacting threads. The first one computes variable values for new launched threads. But the use of a name in next instants may affect its current usage mode and constitutes a backward flow. Therefore the second part of our transition relation updates interacting thread values.

The abstract elements used give, for each thread environment variable and for each binder, a usage mode. These usage modes count the number of times each variable can be used to install an actor. These abstract values denote a range of possible installations: the value \circ means $\llbracket 0, 0 \rrbracket$ and the value \bullet means $\llbracket 0; 1 \rrbracket$. The \top value denotes any other bigger range and invalidates the property. A binder associated to such \top value could potentially be used to bind an actor more than once in the same future configuration.

We define our linearity abstract semantics as follows: we first launch new threads associating to each of their variable a usage mode. If an internal ν binder is defined in the continuation, its value is updated according to its local use. Then the main computation occurs: the backward flow is evaluated, propagating the mode summarizing a variable use back to its origin, *i.e.* the interacting thread variable that defined it.

The abstract domain is presented in two flavors. The first one is the simpler definition, it relies on the definition of the ν binder value computation and of the backward flow. But the soundness of this domain has not proved as is by the author.

Therefore we introduce a second domain, as an extension of the first, that facilitates the soundness proof. However this second formulation necessitates to compute least fixed points inside operational semantics primitives. We conclude with a conjecture linking fixed points of the first domain to fixed points of the second one.

6.2.2 A first abstraction

As before, we first introduce the abstract domain lattice. Then the abstract operational semantics primitives and the operational semantics itself are defined. A last part addresses the difficulties encountered and motivates the second domain definition.

Abstract domain

We introduce the poset \mathbb{N}^b denoting ranges of natural numbers, our usage modes. In this context of linearity checking for CAP, we use the following set:

$$\mathbb{N}^b \triangleq \{\perp, \circ, \bullet, \top\}$$

where \circ denotes the range of natural numbers $\llbracket 0; 0 \rrbracket$ and \bullet denotes the range $\llbracket 0; 1 \rrbracket$. The value \top denotes any other bigger range. In this particular case, \mathbb{N}^b is fitted with a total order \sqsubseteq^b .

$$\perp^b \sqsubseteq^b \circ \sqsubseteq^b \bullet \sqsubseteq^b \top^b$$

We also define the union \sqcup^b as the least upper bound of its arguments and the sum $+^b$ as the symmetric operator such that

$$\left\{ \begin{array}{l} \forall x \in \mathbb{N}^b, \perp^b +^b x = \perp^b, \\ \circ +^b \circ = \circ, \\ \circ +^b \bullet = \bullet, \\ \bullet +^b \bullet = \top^b, \\ \forall x \in \{\circ, \bullet\}, \top^b +^b x = \top^b \end{array} \right.$$

The \sum^b operator is defined as usual using the $+^b$ operator.

We introduce the lattice Nu^{lin} and Env^{lin} build upon \mathbb{N}^b . The first one associates elements of \mathbb{N}^b to CAP address binders in \mathcal{L}_v . The second is similar to the non relational abstraction of thread environments in the control flow abstraction: it associates to each variable of each thread program point an abstract value in \mathbb{N}^b .

$$\text{Nu}^{\text{lin}} \triangleq \mathcal{L}_v \rightarrow \mathbb{N}^b \quad \text{Env}^{\text{lin}} \triangleq \mathcal{L}_p \rightarrow (\mathcal{V} \rightarrow \mathbb{N}^b)$$

Their associated operators, pre-order \sqsubseteq^{Nu} , union \sqcup^{Nu} , bottom element \perp^{Nu} and \sqsubseteq^{Env} , \sqcup^{Env} , \perp^{Env} respectively, are defined as the point-wise extensions of their associated operators in \mathbb{N}^b to discrete maps from \mathcal{L}_v or from $\mathcal{L}_p \times \mathcal{V}$.

The main domain \mathcal{C}^{lin} is pair-wisely defined as the product of Nu^{lin} and Env^{lin} :

$$\mathcal{C}^{\text{lin}} \triangleq \text{Nu}^{\text{lin}} \times \text{Env}^{\text{lin}}$$

The associated operators, pre-order \sqsubseteq^{lin} , union \sqcup^{lin} , bottom element \perp^{lin} are pair-wisely defined using \sqsubseteq^{Nu} , \sqcup^{Nu} , \perp^{Nu} and \sqsubseteq^{Env} , \sqcup^{Env} , \perp^{Env} , respectively.

We do not need to introduce a specific widening operator since our underlying mode domain \mathbb{N}^b satisfies the ascending chain condition (ACC): $\nabla^{\text{lin}} = \sqcup^{\text{lin}}$.

Abstract semantics primitives

COMPUTING BINDER MODE The primitive binder computes for each binder defined in a static continuation the sum of its use in the continuation.

It returns an element of type Nu^{lin} with the new usage modes computed.

Definition 6.5 (binder mode computation) Let $\text{pps}^\# \in \text{Env}^{\text{lin}}$ be an abstract environment. Let $\text{cont} \in \wp(\mathcal{L}_p \times \wp(\mathcal{V} \times \mathcal{L}))$ be a static continuation, associating to each thread to be launched a static environment defining the newly created values. We define the primitive binder as:

$$\text{binder}(\text{cont}, \text{pps}^\#) \triangleq \lambda x. \sum_{(p,v) \in \text{cont}(x)}^b \text{pps}^\#(p)(v)$$

$$\text{with } \text{cont}(x) = \left\{ (p, v) \text{ s.t. } \left\{ \begin{array}{l} (p, E_s) \in \text{cont}, \\ v \in \text{Dom}(E_s) \\ \text{and } E_s(v) = x \end{array} \right\} \right\}.$$

The call of the primitive, on a continuation giving an empty set $\text{cont}(x)$ for all possible x , returns the default value \circ .

FORWARD FLOW The forward flow computation is fairly simple. It just associates a default value to each launched thread. Unreachable threads are then associated to a bottom value when reachable ones have either a default value or a bigger one computed by the backward flow.

The default usage mode is \circ for all thread environment variables except actor addresses which are associated to \bullet .

This simple forward flow does not depend on the already computed value.

Definition 6.6 (static forward flow) Let $\text{cont} \in \wp(\mathcal{L}_p \times \wp(\mathcal{V} \times \mathcal{L}))$ be a static continuation. We define the primitive static_flow such that:

$$\text{static_flow}(\text{cont}) \triangleq \lambda p. \lambda v \in I(p). \left\{ \begin{array}{l} \bullet \text{ when } p \text{ is an actor and } v \text{ its address} \\ \circ \text{ otherwise} \end{array} \right.$$

The forward flow computation step can then be defined. It updates the abstract element with the simple forward flow computed and summarizes the use of internal binders launched.

Definition 6.7 (forward flow computation) Let $(b^\#, \text{pps}^\#) \in \mathcal{C}^{\text{lin}}$ be an abstract configuration and $\text{cont} \in \wp(\mathcal{L}_p \times \wp(\mathcal{V} \times \mathcal{L}))$ be a static continuation. We introduce the primitive fflow :

$$\text{fflow}(\text{cont}, (b^\#, \text{pps}^\#)) \triangleq (b'^\#, \text{pps}'^\#)$$

where

- $\text{pps}'^\# = \text{static_flow}(\text{cont}) \sqcup^{\text{Env}} \text{pps}^\#;$
- $b'^\# = \text{binder}(\text{cont}, \text{pps}'^\#) \sqcup^{\text{Nu}} b^\#.$

DEPENDENCIES In order to facilitate the latter binding between variables of interacting threads and variables in launched thread environments, we introduce the primitive dependencies. It extracts links between variables using the non standard semantics elements: the formal rule and the partial interactions associated to interacting threads. These links come either from the value passing computed during the transition, *i.e.* message argument and ζ variable, or it binds variables of launched threads to the same variables of the static actor or behavior branch thread. This second flow denotes the natural binding of variables.

Definition 6.8 (dependencies) *Let \mathcal{R} be a CAP formal rule of the non standard semantics. Let $(p^k)_k$ be a sequence of interacting thread program points and $(pi^k)_k$ be their associated partial interactions. We recall that in the CAP non standard encoding the continuation is only defined in the first partial interaction denoting either a static actor or a behavior branch, depending on the matched rule.*

$$\begin{aligned} \text{dependencies}(\mathcal{R}, (p^k)_k, (pi^k)_k) &\triangleq \{(p, v), (p', v')\} \\ &\text{s.t. } \exists E_s \in \wp(\mathcal{V} \times \mathcal{L}_v), (p, E_s) \in \text{continuation}^1, v \in I(p) \setminus \text{Dom}(E_s), \\ &\left\{ \begin{array}{l} \text{either } p' = pi^i, v = \text{param}_j^1, \text{ and } v' = \text{bound}_i^1 \text{ when } (Y_j^1 \leftarrow X_i^1) \in v_passing \\ \text{either } p' = p^1 \text{ and } v = v' \text{ otherwise} \end{array} \right. \end{aligned}$$

where

- $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_passing)$,
- $(pi^k)_k = (s^k, (\text{param}_i^k), (\text{bound}_i^k), \text{continuation}^k)_k$.

BACKWARD FLOW The backward flow computation takes the dependencies computed between interacting thread environment variables and launched thread environment variables. For each variable, it computes the sum of its use in the continuation and updates the value associated to the source variable in the interacting thread abstract environment.

Definition 6.9 (backward flow computation) *Let $\text{dep} \in \wp((\mathcal{L}_p \times \mathcal{V}) \times (\mathcal{L}_p \times \mathcal{V}))$ be a set of dependencies between interacting thread variables and launched thread ones. Let $(b^\#, \text{pps}^\#) \in \mathcal{C}^{\text{lin}}$ be an abstract element. We define the primitive bflow as follows.*

$$\text{bflow}(\text{dep}, (b^\#, \text{pps}^\#)) \triangleq b^\#, \lambda p. \lambda v \in I(p). \left(\text{pps}^\#(p)(v) \sqcup^b \sum_{(p', v') \in \text{succ}(p, v)}^b \text{pps}^\#(p')(v') \right)$$

where $\text{succ}(p, v) = \{(p', v') \text{ s.t. } ((p', v'), (p, v)) \in \text{dep}\}$.

UPDATE INITIAL THREADS ABSTRACTION Finally the last primitive allows to update the value of initial threads. In the original framework, the property obtained in the initial configuration is defined by the $C_0^\#$ abstract value but does not evolve with transitions. For instance, in the control flow abstraction, the initial threads are initially fully defined and no future thread will ever be launched with these initial program points.

In our abstract domain, the backward flow could update the abstract values associated to initial threads. We need to ensure that the initial term init_s , as defined by our CAP extraction function satisfies the linearity property considering the computed abstract element.

Definition 6.10 (initial binder updating) Let $(p^k)_k$ be a set of interacting thread program points and $(b^\#, \text{pps}^\#) \in \mathcal{C}^{\text{lin}}$ be an abstract element. We define the primitive init as follows.

$$\text{init}((p^k), (b^\#, \text{pps}^\#)) \triangleq \begin{cases} b^\# \sqcup^{\text{Nu}} \text{binder}(\text{init}_s, \text{pps}^\#), \text{pps}^\# & \text{when } \exists p \in P, \\ & \exists E_s \in \wp(\mathcal{V} \times \mathcal{L}) \text{ s.t. } (p, E_s) \in \text{init}_s \\ b^\#, \text{pps}^\# & \text{otherwise} \end{cases}$$

Abstract operational semantics

We are now able to define the abstract operational semantics.

INITIAL STATE The initial state is defined using the fflow primitive: it associates default values to initial threads and their sum to the initial binders.

Definition 6.11 (initial abstract state) Let init_s be the initial static configuration. We define the initial abstract element C_0^{lin} as follows:

$$C_0^{\text{lin}} \triangleq \text{fflow}(\text{init}_s, \perp^{\text{lin}})$$

ABSTRACT TRANSITION The abstract transition can then be defined. Its formal definition is given in Figure 6.2.

Let us just recall the principal steps:

- in this domain, the transition is only constrained by syntactic conditions such as the message label or arity. It does not consider the compatibility constraints of the considered formal rule;
- launched threads are initialized with a default value, • for actor addresses and ◦ for other variables;
- newly defined binders are associated to the sum of their use in the launched continuation;

Figure 6.2 Abstract operational semantics for first abstraction.

Let $C^\# \in \mathcal{C}^{\text{lin}}$ be an abstract configuration. Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, \text{v_passing})$ be a formal rule. Let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program point labels and $(\text{pi}^k)_{1 \leq k \leq n} = (s^k, (\text{par}^k), (\text{bd}^k), \text{cont}^k)$ be a tuple of partial interactions.

We define the abstract transition involving the rule \mathcal{R} , the program points $(p^k)_k$ and their associated partial interactions $(\text{pi}^k)_k$ as:

$$C^\# \xrightarrow{(\mathcal{R}, (p^k)_k, (\text{pi}^k)_k)}^{\text{lin}} \text{fd} \cup \text{bd} \cup \text{init}$$

where

- $\text{fd} = \text{fflow}(\text{cont}^1, C^\#)$;
 - $\text{dep} = \text{dependencies}(\mathcal{R}, (p^k)_k, (\text{pi}^k)_k)$;
 - $\text{bd} = \text{bflow}(\text{dep}, \text{fd})$;
 - $\text{init} = \text{init}((p^k)_k, \text{bd})$.
-

- a backward flow is computed, updating each interacting thread environment depending on the use of their variable in the launched continuation;
- finally the initial threads and binders are updated in order to detect linearity failure on initial binder addresses.

Concretization and soundness

The difficulty in the presented domain is to characterize a useable concretization map. In practice, the concretization is used to guarantee the soundness of the domain by construction but is not used when computing transitions. It is also useful to extract information from the abstract element obtained at the end of the fixed point computation.

In this particular case of linearity checking, the motivation was to compute an usage mode for each binder and variable use. The idea behind was to be able, considering an abstract element over-approximating sets of configurations, to affirm that all the related concrete configurations were linear or not, depending on the binder usage mode. If a binder usage mode is \perp^b , the binder is never used in reached threads; if the binder is \circ , no actor is ever installed on this binder addresses; if the binder is \bullet , at most one actor is installed in reachable configurations; and finally if the binder mode is \top^b we cannot ascertain that related configurations are linear ones.

Such a concretization is monotonic and would satisfy the soundness assumption required by our framework. However it is not applicable here. In fact when computing a transition that breaks the linearity, the linearity failure will be detected, associating a \top^b to the associated binder, only latter in the fixed point computation. It requires to propagate the mode through the backward flow computations until the continuation launching where linearity is broken in practice (two threads containing the same variable x with both a value \bullet in a continuation). And again it propagates the \top^b mode back to the binder definition.

Conjecture 6.12 *Considering an adequate monotonic concretization γ_{lin} relating post fixed point abstract elements to sets of configurations, the abstraction $(\mathcal{C}^{\text{lin}}, \sqsubseteq^{\text{lin}}, \sqcup^{\text{lin}}, \perp^{\text{lin}}, \gamma^{\text{lin}}, C_0^{\text{lin}}, \rightarrow^{\text{lin}}, \nabla^{\text{lin}})$ is a sound abstraction with respect to the Definition 3.3.*

6.2.3 A second abstraction

In order to address this soundness requirement, we introduce a modification of the first domain in which each computation of an abstract transition reduces to a fixed point computation propagating uses of variables back to their defining binder.

We extend the abstract domain with dependency relations between thread variables and modify the backward computation in order to introduce the fixed point computation.

Then each abstract element can now be related to concrete configurations through a monotonic concretization map.

Abstract domain

We introduce the set Dep that allows us to store each dependency relation that arises between interacting thread environment variables and launched thread ones. These dependencies were the ones used in the backward flow primitive and produced by the dependencies primitive.

$$\text{Dep}^{\text{lin}} \triangleq \wp((\mathcal{L}_p \times \mathcal{V}) \times (\mathcal{L}_p \times \mathcal{V}))$$

This set is associated to usual set operators: the pre-order \sqsubseteq^{Dep} defined as the set inclusion \subseteq , the union operator \sqcup^{Dep} as the set union \cup and the bottom element \perp^{Dep} as the empty set \emptyset .

The main domain $\mathcal{C}^{\text{lin}'}$ is now defined as follows:

$$\mathcal{C}^{\text{lin}'} \triangleq (\text{Nu}^{\text{lin}} \times \text{Env}^{\text{lin}}) \times \text{Dep}^{\text{lin}}$$

Operators are extended on this product.

Concretization

The concretization $\gamma^{\text{lin}'}$ is the one that was defined informally above on fixed points of the first abstract domain.

It is quite complex since each part of the abstract element addresses specific properties:

- the binder part specifies the usage of each address, linear or not;
- the abstract environment gives the over-approximation of variable values usage;
- the dependency part reflects all the links between environment variables induced by the value passing of computed transitions.

$$\left\{ (u, C) \left| \begin{array}{l} \forall (b, \text{id}) \in \mathcal{L}_v \times \mathcal{M}, C_{|(b, \text{id})} : \\ \left\{ \begin{array}{l} \text{is empty when } b^\#(b) = \perp^b; \\ \text{contains only messages when } b^\#(b) = \circ; \\ \text{contains at most one actor when } b^\#(b) = \bullet. \end{array} \right\} \end{array} \right. \right\} \\ \subseteq \gamma^{\text{binder}}(b^\#)$$

The concretization devoted to the binder approximation restricts the use of each address in over-approximated configurations. Each represented configuration is restricted for each address depending on the usage mode of the address binder. A \perp^b value forbids the use of the address to bind threads. A \circ value or a \bullet one denotes a linear configuration with respectively no messages or at most one actor. Finally, the \top^b value does not constrain at all the associated sub-configuration.

$$\left\{ (u, C) \left| \begin{array}{l} \forall (p, \text{id}, E) \in C, \forall v \in I(p), \text{pps}^\#(p)(v) \neq \perp^b \\ \forall \lambda \text{ s.t. } u = v.\lambda.v', C_0 \xrightarrow{v} C_i \xrightarrow{\lambda} C_i \cup \text{newt} \setminus \text{removed} \xrightarrow{v'} C \\ \forall (p', \text{id}', E') \in \text{newt}, \forall x' \in I(p), \\ \exists ((p_o, x_o), (p', x')) \in \text{dep}^\# \text{ s.t. } (p_o, \text{id}, E) \in \lambda \wedge E'(x') = E(x) \\ \forall (p, \text{id}, E) \text{ actor thread} \in C, \\ \forall (p', x') \text{ s.t. } (p, \text{address_var}(p)) \rightsquigarrow^+ (p', x'), \text{pps}^\#(p')(x') \sqsupseteq \bullet \end{array} \right. \right\} \\ \subseteq \gamma^{\text{modes}}(\text{pps}^\#, \text{dep}^\#)$$

where $(p_1, v_1) \rightsquigarrow (p_2, v_2) \equiv ((p_1, v_1), (p_2, v_2)) \in \text{dep}^\#$.

This concretization is more complex as it expresses the relationships between variable values that were implicit in the first abstraction, due to value passing.

The first part expresses that no bottom thread can be present within C .

The second part makes explicit the sequences of transitions that occurred: each created thread variable in the past of the configuration C has values that came from interacting threads that created it.

Finally the third part constrains all thread environment variables that can be used to bind an actor. Any variable that can later be used that way has an usage mode greater than \bullet .

Both last parts are essential in the latter proof to guarantee that non linearity is detected through abstract transition computation.

Then the main concretization is defined as follows:

$$\gamma^{\text{lin}'}((b^\#, \text{pps}^\#), \text{dep}^\#) \triangleq \gamma^{\text{binder}}(b^\#) \cap \gamma^{\text{modes}}(\text{pps}^\#, \text{dep}^\#)$$

Abstract semantics primitives

The abstract primitives are modified to make the fixed point computation explicit and to update the value passing dependencies between variables.

BACKWARD FLOW REVISITED: PROPAGATING MODES The backward flow is modified in order to propagate the backward mode updating to the binder defining values.

In the following, we denote by $\text{cont}(p)$ the static continuation where the thread at program point p is defined. It can be syntactically determined. Initial threads are associated to init_s , by this function.

We first define the single step of backward computation. It takes all the dependencies in $\text{dep}^\#$ and propagates back the modes along these dependencies. All internal ν binders are also updated using the binder primitive.

Definition 6.13 (one step backward) *Let $((b^\#, \text{pps}^\#), \text{dep}^\#) \in \mathcal{C}^{\text{lin}'}$ be an abstract element. The single step of backward computation is defined as follows:*

$$\text{Step}((b^\#, \text{pps}^\#), \text{dep}^\#) \triangleq (b'^\#, \text{pps}'^\#)$$

where

- $(b'^\#, \text{pps}'^\#) = \text{bflow}(\text{dep}^\#, (b^\#, \text{pps}^\#));$
- $b'^\# = b^\# \sqcup^{\text{Nu}} \bigsqcup_{\text{cont}(p) \text{ s.t. } p \in \mathcal{L}_p} \text{binder}(\text{cont}(p), \text{pps}^\#).$

Now the propagation can be expressed as the fixed point of this single step primitive.

Definition 6.14 (modes propagation) Let $(C^\#, \text{dep}^\#) \in \mathcal{C}^{\text{lin}'}$ be an abstract element. We define the backward flow computation using the primitive propagate expressed as the least fixed point of the Step primitive:

$$\text{propagate}(C^\#, \text{dep}^\#) \triangleq \text{lfp}_{\perp_{\text{lin}}} (\lambda X. C^\# \sqcup \text{Step}(X, \text{dep}^\#))$$

UPDATE INITIAL THREADS The previous function `init` is no longer necessary since the computation of binder modes is done within the `Step` function. Program points of initial threads are associated to the set init_s of static threads by the function `cont` and the binder modes of these initial threads updated.

Abstract operational semantics

The abstract operational primitive is defined as before.

INITIAL STATE The initial state $C_0^{\text{lin}'}$ considers an initial empty set of dependencies.

Definition 6.15 (initial abstract state)

$$C_0^{\text{lin}'} = (C_0^{\text{lin}}, \emptyset)$$

OPERATIONAL SEMANTICS The operational semantics is described in Figure 6.3. As in the previous abstraction, first a forward flow is computed, launching new threads. Then the backward flow computation propagates back the mode to the possible original binder, computing the sum of the usages at each intermediate computation step.

Soundness

We now state the soundness of this second domain and conjecture the soundness of the first abstraction.

Theorem 6.16 *The abstraction $(\mathcal{C}^{\text{lin}'}, \sqsubseteq^{\text{lin}'}, \sqcup^{\text{lin}'}, \perp^{\text{lin}'}, \gamma^{\text{lin}'}, C_0^{\text{lin}'}, \rightarrow^{\text{lin}'}, \nabla^{\text{lin}'})$ is a sound abstraction with respect to the Definition 3.3.*

Proof 6.17 *The proof can be found in A.4.*

We now state the conjecture relating fixed points of both abstractions.

Conjecture 6.18 *Considering a non standard initial term C_0 , and an associated finite set of transition labels, the fixed points of the collecting semantics expressed in the domains \mathcal{C}^{lin} and $\mathcal{C}^{\text{lin}'}$ are identical.*

Proof 6.19 (proof sketch) *The dependencies accumulated in the $\mathcal{C}^{\text{lin}'}$ domain that drive the backward fixed point computation are the ones that are used in each single abstract transition in the domain \mathcal{C}^{lin} .*

Figure 6.3 Abstract operational semantics for second abstraction.

Let $(C^\#, \text{dep}^\#) \in \mathcal{C}^{\text{lin}}$ be an abstract configuration. Let $\mathcal{R} = (n, \text{components}, \text{compatibility}, \text{v_passing})$ be a formal rule. Let $(p^k)_{1 \leq k \leq n} \in \mathcal{L}_p$ be a tuple of program point labels and $(\text{pi}^k)_{1 \leq k \leq n} = (s^k, (\text{par}^k), (\text{bd}^k), \text{cont}^k)$ be a tuple of partial interactions.

We define the abstract transition involving the rule \mathcal{R} , the program points $(p^k)_k$ and their associated partial interactions $(\text{pi}^k)_k$:

$$(C^\#, \text{dep}^\#) \xrightarrow{(p^k)_k}^{\text{lin}'} (\text{fd} \cup \text{bd}, \text{dep}^{\#})$$

where

- $\text{fd} = \text{fflow}(\text{cont}^1, C^\#)$;
- $\text{dep}^{\#} = \text{dep}^\# \sqcup^{\text{dep}} \text{dependencies}((p^k)_k, \text{cont})$;
- $\text{bd} = \text{propagate}((\text{fd}, \text{dep}^{\#}))$.

6.3 EXAMPLE ANALYSIS

The Figure 6.4 gives a possible sequence of abstract transition computations leading to a \top^b value for the binder at program point γ in a minimal number of steps.

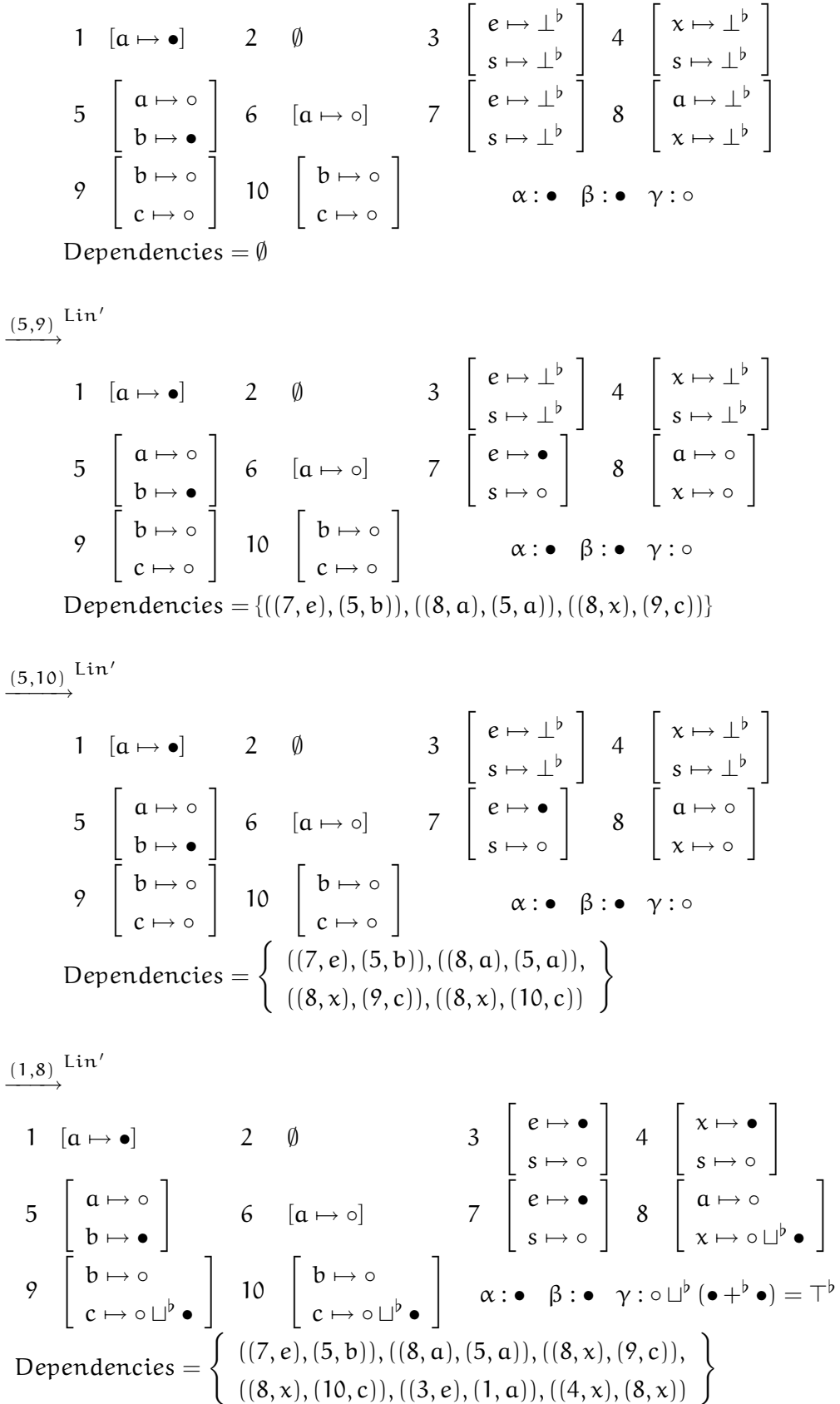
The first two steps launch the reached threads and accumulate created dependencies between thread variables. In particular, they express the relation between the variables c of threads at program points 9 and 10 and the variable x of threads at program point 8. Until then, there is no valuable backward flow computation. The last transition presented, (1, 8), launches the thread at program point 4 using the variable x to bind an actor. The backward flow computation with its fixed point definition propagates this value to the variable x at program point 8 and then at the variables c at program points 9 and 10.

The updating of the associated binder detects that the variable c is used twice with a \bullet mode and goes to \top^b .

The first abstract domain without the local least fixed point definition necessitates the computation of the abstract transitions (5, 9) and (5, 10) to propagate the \bullet value back to the threads at 9 and 10 and detects then the linearity failure.

6.4 RELATED WORKS

Linearity has been studied using different approaches in the context of concurrency.

Figure 6.4 Linearity: abstract properties computation.

The first works related to the current analysis are the ones of COLAÇO *et al.* in [26] where the property was checked in a type inference based approach using a HM(X) type system. This type system deals with behavior passing computing a usage mode for each variable. The constraints accumulated when typing a term denote causality relations between sub-processes.

This type system was the only one proposed by COLAÇO that handles CAP behavior passing capability, but constraining the information contained in accumulated constraints. Our approach seems more promising considering the precision of our control flow analysis, driving abstract transitions, and considering the way we propagate usage mode. In our approach to handle behavior passing, we do need to restrict the number of modes and could express more complex abstractions such as intervals to count numbers of actors bound to the same address. Furthermore, the internal free variables in a behavior are not automatically associated to a \top^b value as in COLAÇO type system.

Other works address a similar property in a more or less different contexts. In [89], the authors proposed a method for inferring the maximum usage of channels, both for input and output, by applying an effect system designed for Concurrent ML [91]. This analysis handles higher-order functional terms but only first order ones for the concurrent part. Furthermore it widens recursive receptions which often lead to ∞ usages.

The work [77] addresses the problem of linear channels in π -calculus. A channel is linear when there is at most one reader and one sender. However the proposed type system is not fitted with an inference algorithm. The multiplicity part of this type system was inspired by the analysis presented in [76] which has been later continued in [64]. It is devoted to counting receptions and does not deal with channels as first class values. Their types carry information about the sequences of actions with a multiplicity, but this could lead to an explosion of the size of types. Another problem is that their type system cannot express properties that deal with the dynamic creation of channel names.

All these approaches are mainly type system based and follow a denotational approach of static analysis. They lack the precision we obtained in a more operational approach using the powerfulness of abstract interpretation to reach an over-approximation in reasonable time. Furthermore no one of them, except the work of COLAÇO, analyzes such properties in a high-order context.

In a more theoretical view, linearity has been studied in [95] comparing the relative expressivity of the semi-persistent flavors of the π -calculus, considering replicated input or output, or in contrary the flavor with no replications.

6.5 DISCUSSION

The proposed domains are precise enough to ensure the linearity as defined in CAP. However we advocate that the different proposed approaches, the one based on computed properties from the control flow approximation and the

occurrence counting one with the abstract domain devoted to the usage of each channel could be combined to obtain better results.

Considering the proposed abstract domain as well as the difficulties encountered when expressing such a backward flow in a sound manner and in the original approach of a least fixed point computation, we target to express such a property as a greatest fixed point. The least fixed point of the control flow abstraction could be used to over-approximate reachable program points with the set of possibly associated binders. Such reachable threads should be associated to \top^b values and backward flow should be computed as intersection of the old value and the one inferred by an abstract transition computation. We believe that this formulation could lead to a direct soundness proof, without the need for a local least fixed point.

Finally the domains proposed here in a CAP context could easily be adapted in order to apply them to more widespread calculi, such as the π -calculus. The simple \mathbb{N}^b domain is used to approximate ranges of usage, it could be replaced with intervals or even relational abstractions between different channel usages. The abstract semantics could also be adapted to count the number of emissions on a channel as well.

We now focus on our second property: the orphan-freeness checking. As briefly presented above in the introduction chapter, orphan messages can appear in asynchronous calculi especially in the so-called calculi *with non uniform interfaces*, where the capacity to handle messages may vary depending on previously computed transitions. In CAP, orphans are messages that are sent to an address that will not be able to ever handle them.

This property is inherently complex since it necessitates to build a representation of the different sequences of interfaces associated to each address value as well as the set of messages available to them at any time. By essence, process calculi do not facilitate such a representation due to the huge number of different possible computation traces. Furthermore, in our case, the CAP behavior passing mechanism is also a gap to bridge in order to be able to get accurate results.

In this chapter, we propose an analysis allowing to check orphan-freeness. The Section 7.1 details the problematics, gives the property formalization and illustrates it on an example. The Section 7.2 gives then an overview of the proposed property checking, reducing it to a reachability problem over *Vector Addition Systems with States* (VASS). In Section 7.3, we propose an abstract domain representing sequences of interfaces as a VASS. In Section 7.4 we rely on the computed abstract element in order to effectively check the property. We also discuss soundness arguments of our approach. Finally the Section 7.5 describes the analysis of an example while Section 7.6 gives a final discussion and considers related works.

7.1 PROBLEMATICS

We are interested in detecting orphan messages in our actor process calculus, CAP. In the actor model, as in most asynchronous process calculi, characterizing and detecting messages that are sent in some execution but won't be handled ever after is a pregnant concern. In this context, an orphan message is intuitively such a sent message that will not have a chance of ever being consumed. In fact, the real property we are interested in is the absence of orphans: the orphan freeness property.

Such property only make sense in valid CAP terms, *i.e.* in linear ones. In all this chapter, we consider linear terms. The previous chapter, Chapter 6, addressed this linearity property analysis.

SAFETY ORPHANS VS. LIVENESS ORPHANS However several flavors of orphan messages can be characterized. A first one may consider messages that will never be consumed, because they do not appear in the future behavior branches of the target actor, we call them *safety orphans*. A second characterization, more difficult to deal with, is the notion of *liveness orphans*. It restricts the preceding definition, considering actor stuck. A message which is not a safety orphan, because the future of its target actor contains a behavior branch with its own label, could be a liveness orphan if its target actor is stuck and is waiting for another message to get the appropriate behavior.

In the following we address the freeness checking of both kind of orphans. The absence of safety orphans is first ensured. Then we verify that a system with non empty mailboxes is not stuck.

GUARANTEEING A HIGH-LEVEL PROPERTY Orphan freeness checking is indeed a high level property which necessitates non trivial analyses. As it is more detailed in the next sections, orphan freeness is a temporal property that is expressed on maximal traces. In practice, it is checked in any reachable state. All maximal traces computable from such states must exhibit the capability to handle all messages present herein. In order to provide an analysis able to deal with orphan freeness, one has to deal with sequences of available interfaces for each address, or at least find a way to model it. Moreover, CAP behavior passing ability renders the framework complex.

7.1.1 Definitions

As said above, ensuring the property requires to represent interfaces associated to an address. We also need to represent the messages available at each step of the computation. We now introduce the necessary definitions.

MAILBOXES All messages available in a given configuration can be partitioned among their target address. We denote by mailbox the element of such partition associated to a given address. In fact we only count the number of messages associated to the same label.

Definition 7.1 (Mailbox) For an actor name a in a given configuration C , we denote by $\text{Mailbox}(a, C)$ the finite multiset of message labels sent to a in C .

FUTURE INTERFACES As the property has to be checked in any reachable state, we can flatten the sequence of future interfaces available in a given configuration while representing future behaviors associated to an address.

Definition 7.2 (Future interface) For an actor name a in a given configuration C , we define $\text{Future}(a, C)$, the future interface of a , as the finite set of message labels it

can handle in some behavior that it may assume in some execution path drawn from C , including the current behavior if any. Formally:

$$\text{Future}(a, C) \triangleq \{m \mid \exists C' : \nu a, C \rightarrow^* \nu a, C' \wedge C' \equiv a \triangleright [m(\dots) = \dots] \parallel \dots\}.$$

This previous definition could be further refined, as two slightly different notions of what it means for an interface to be able to handle some message could be adopted. First, we may stick to the definition and consider the static set of branches that are attached to some behavior, or second we may only include branches that may take part in a transition of some execution, that is we may rule out some dead behavior code.

Here, we take the second option, as the abstract interpretation phase aims at computing an approximation of the set of reachable configurations, striving to avoid exploring dead code whenever possible.

ORPHAN MESSAGES We can then define what orphan messages are. We recall that we consider here the first flavor presented above: *safety orphans*. Orphans can then be defined locally, on a configuration, considering the flattened sequence of future interfaces.

Definition 7.3 (Orphan message) For a given actor name a and message $a \triangleleft m(\dots)$ in its mailbox in a given configuration C , m is declared orphan if not present in the future interfaces of a in C .

$$\text{Orphan}(m, a, C) \triangleq m \notin \text{Future}(a, C) \wedge C \equiv \nu a, a \triangleleft m(\tilde{x}) \parallel \dots$$

ORPHAN-FREE CONFIGURATIONS Finally the main definition ensures that all reachable configurations are orphan free. A faulty message can then *validate* the property for initial states but then invalidates it when reaching a configuration where not future handles it.

Definition 7.4 (Orphan-free configuration) A configuration C is orphan-free if it does not contain orphan messages, and its possible successor configurations are all orphan-free. Formally:

$$\text{OrphanFree}(C) \triangleq \forall a, m, C' : C \rightarrow^* C' \Rightarrow \neg \text{Orphan}(m, a, C').$$

This property can also be expressed as a CTL-like formula:

$$\text{OrphanFree}(C) \triangleq \forall \square C \equiv \nu a, a \triangleleft m(\tilde{x}) \parallel \dots \implies \exists \diamond C \equiv \nu a, a \triangleright [m(\tilde{y}) = \dots] \parallel \dots$$

Here we state an interesting property of orphan-free configurations, under a particular fairness assumption. This assumption, which is usual in the actor model, can be easily enforced using a queue-based implementation of pending messages for instance. Note that this property does not prevent waiting queues from arbitrarily growing.

Property 7.5 (Finite pending time) *Provided old messages will eventually be handled (if possible) before newer ones, then messages cannot remain pending forever in a mailbox.*

This property guarantees us a fair receiving in case of infinite behaviors. However as our verification methodology is static analysis, we build all possible interleavings and do not lose cases.

A second fairness property forbids bad infinite behaviors.

Property 7.6 (Finite firing time) *Provided necessary messages exist (old ones or new ones), behavior branches that could be infinitely often be used will eventually be.*

A last property to introduce allows to define liveness orphan free systems.

Property 7.7 (Non stuck actors) *An actor, associated to a non empty mailbox, is non stuck provided it will be able to compute a transition. We rely on the non standard semantics in order to express the actor consumption.*

$$\begin{aligned} \text{NonStuck}(C) &\triangleq \\ &\forall(p, \text{id}, E) \text{ such that } p \in \mathcal{L}_a \text{ and } \text{add} = E[\text{address_var}(p)], \\ &\forall \square(p, \text{id}, E) \in C \wedge \text{Mailbox}(\text{add}, C) \neq \emptyset \implies \forall \diamond(p, \text{id}, E) \notin C \end{aligned}$$

We can now express a strong property expressing message consumptions using both the liveness orphan freeness and the finite pending time property. We rely on the non standard encoding of CAP as it allows to differentiate recursive instance of the same program point. A similar definition in the standard semantics would require a time stamp on messages to differentiate them.

Property 7.8 (Message consumption) *Let C_0 be an orphan-free non standard configuration. We have the following property, considering the finite pending time property, the finite firing time property and non stuck actors:*

$$\forall(p, \text{id}, E) \text{ such that } p \in \mathcal{L}_m, \forall \square(p, \text{id}, E) \in C \implies \forall \diamond(p, \text{id}, E) \notin C$$

Finally, it turns out that we need to compute, for a given actor a , the set of configurations C in which it may occur (with its associated mailboxes) and the set of its future interfaces $\text{Future}(a, C)$ for each such configuration $C \in \mathcal{C}$. We need then to compute, for all message label m , an over-approximation of its reachable states where the occurrence of messages labeled m is strictly positive; and an under-approximation of the reachable states that consume such kind of messages. Lastly, they are compared by inclusion to ensure their consumption. This step ensures the safety orphan freeness.

Then we ensure that each actor is able to compute a transition when its mailbox is non empty.

7.1.2 Examples

Let us illustrate these definitions on some simple examples.

Example 8 (Linear cell) *The linear cell presented in Example 7.1a is the basis of these examples. It is composed of a single actor on address z . This actor is able to receive a put message with one argument. When receiving such a message, it assumes another behavior allowing it to receive a get message with one argument. Handling such a get message produces a reply message labeled rep addressed to the argument of the get message with argument the one of the put message. When receiving the second message, the one labeled get , it comes back to its initial behavior. It is able to receive a put message and has no information about past transitions (i.e. values of past handled put and get messages).*

Example 9 (S_1 : Linear cell with a single request Put-Get) *The first system, presented in Example 7.1b, is composed of a single linear cell on address a and two initial messages put and get sent to address a . It has a single finite possible execution. It first handles the put labeled message, then the get labeled one and produces the reply message, labeled rep , while assuming its initial behavior.*

The Figure 7.1a gives a representation of its finite single trace. Each node represents a configuration with its associated set of messages. A transition denotes a message consumption building a new configuration.

One can then characterize Future for each of the three reachable configurations.

$$\text{Future}(a, C_0) = \{put, get\}$$

$$\text{Future}(a, C_1) = \{put, get\}$$

$$\text{Future}(a, C_2) = \{put, get\}$$

All messages in C_0 and in C_1 are included in the associated future interface but the last message labeled rep is not included in C_2 .

$$rep \notin \text{Future}(a, C_2)$$

It is orphan and therefore the system is not orphan free.

Example 10 (S_2 : Linear cell with a Put-Get requests server) *The second system, presented in Example 7.1c, is composed of two actors on two separate addresses. On address a , the linear cell receives messages sent by the actor on address b . This second actor produces one message put and one message get both sent to a when receiving a message rep . This system can loop forever once the second actor is initialized. An initial message labeled rep is then sent to b in order to initiate the computation.*

The Figure 7.1b gives a representation of its infinite single trace. One can then characterize Future for each reachable configurations.

$$\forall C', \text{ s.t. } C \rightarrow^* C', \begin{cases} \text{Future}(a, C') = \{put, get\} \\ \text{Future}(b, C') = \{rep\} \end{cases}$$

Example 7.1 Systems based on linear cell.

(a) Linear cell

$$\begin{aligned}
 & z \triangleright^1 [\text{put}^2(x) = \zeta(e, s)(\\
 \text{LC}(z) \triangleq & \quad e \triangleright^3 [\text{get}^4(y) = \zeta(e', s')(e' \triangleright^5 s \\
 & \quad \quad \quad \parallel y \triangleleft^6 \text{rep}(x) \\
 & \quad \quad \quad)]]]
 \end{aligned}$$

(b) A first system

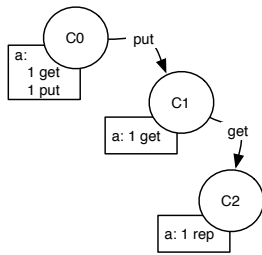
$$S_2 \triangleq \nu a, \nu x, \text{LC}(a) \parallel a \triangleleft \text{put}(x) \parallel a \triangleleft \text{get}(a)$$

(c) A second system

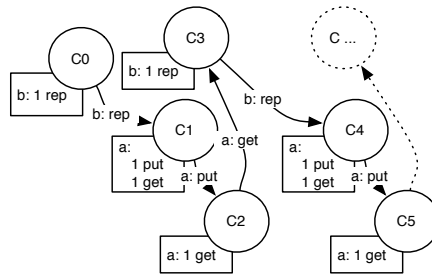
$$S_1 \triangleq \nu a, \nu b, \left\{ \begin{array}{l} \text{LC} \\ \parallel b \triangleright [\text{rep}(v) = \zeta(e, s)(e \triangleright s \parallel \nu x, a \triangleleft \text{put}(x) \parallel a \triangleleft \text{get}(e))] \\ \parallel b \triangleleft \text{rep}(b) \end{array} \right.$$

Figure 7.1 Example traces with mailboxes.

(a) S_1 trace



(b) S_2 trace



At every step of the computation, all mailboxes for address a contain at most one message labeled put and one message labeled get when mailboxes for address b contain at most one message labeled rep . The condition is satisfied: $\forall x \in \mathcal{N}, \forall C \in \mathcal{C}, \{m \mid (m, \text{occ}) \in \text{Mailbox}(x, C)\} \subseteq \text{Future}(x, C)$. The system S_2 is orphan-free.

Furthermore, at each step, when the mailbox of an actor is non empty, the actor will be able to handle a message in the future. An actor on address a with a non empty mailbox is always able to handle a message (in configurations C_{1+3k} and C_{2+3k}). An actor on address b with a non empty mailbox has a similar property (in configurations C_{3k}).

7.2 ROADMAP TO ORPHAN FREENESS CHECKING

7.2.1 *Observation*

If we have a look on our orphan-free S_2 system, we can observe particularities of such system and try to derive a systematic way to check orphan freeness.

Example 11 (System S_2 (continued)) *As presented above and clearly visible in its trace (see 7.1b), this infinite system behave in a cyclic way. We can observe sub-traces similar to the one of S_1 . The only difference is a computation allowing to reenter the initial state.*

So even if the system is infinite, we can detect three CAP terms ¹ defining three interfaces associated to addresses.

The first two interfaces define the linear cell behavior. Let us denote by interface A and B the two interfaces receiving put and get respectively. The third one, C, is the only interface of the PutGet server, with the capability to handle rep messages.

All messages in the initial term and in the reachable ones only contain messages labeled in the set {put, get, rep}.

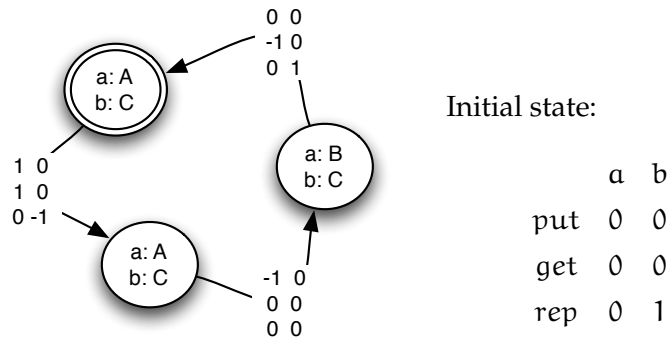
Then this system, either its mailboxes and interfaces, can be represented as a vector system as in Figure 7.2. A vector denotes the multiset of labeled messages associated to each address. Here we have two addresses and three possible messages, hence a 6-dimensional vector. An automaton denotes the possible transitions. Each state of the automaton associates an interface to each address. Finally transitions model mailboxes evolutions through a vector computation. For example, the transition from the state (a : B, b : C) to the state (a : A, b : C) consumes one message get on a and produces one message rep on b.

Such a representation, when it exists, gives us a way to represent all reachable states with their associated mailboxes.

ORPHAN FREENESS AS A COVERABILITY PROBLEM Orphan freeness checking can then be expressed as a reachability problem. One has to ensure that for each non empty mailbox there exists a transition consuming each message in the reachability set of this state (cf. Definition 7.4).

STUCK ACTOR FREENESS AS A COVERABILITY PROBLEM In a similar way, the freeness of stuck actor can be expressed on this model. One has to ensure that for each state with an actor associated to a non empty mailbox, there exists a transition that consumes the actor. (cf. Definition 7.7).

¹ Note that this is not true anymore when considering its encoding into the non standard form. Identity markers of threads grow at each computation.

Figure 7.2 System S_2 : an (infinite) ideal case.

The objective of the encoding is to check that each existing message label in a configuration can be consumed in the future of the term and that each actor is non stuck. However this encoding of our property has to be relativized as is. First the ideal case presented above as a finite set of name binders (only two) which are syntactically known. In case of internal ν operator and recursion, the number of dimensions will grow and would forbid such a finite representation. Another big problem is the finiteness of our system. In our ideal case, the infinite system has a finite representation. But if every interface is new, parametrized by a new value for example, then we will not be able to represent it as a finite automaton.

7.2.2 Vector Addition System with States and their properties

The previous encoding into vector systems is indeed a quite famous target model. In [67], KARP and MILLER analyze this mathematical structure. They argue that such systems arise in many independent area such as the word problem for communication semigroups, the theory of bounded context-free languages and the theory of uniform recurrence equations. They call it *vector addition system* (VAS). A VAS state is a positive vector of fixed dimension. A VAS specification allows to build new states from the initial one, considering only positive vectors.

As we try to count occurrences of message labels, VAS is a natural formalism. In [67], the authors also analyze some decidability properties and propose encodings of several problem. We now recall some of these definitions and properties that allow us to ensure orphan freeness.

Definition 7.9 (VAS and reachability set) A r -dimensional VAS is a pair $\mathcal{W} = (d, W)$ in which d is an r -dimensional vector of nonnegative integers, and W is a finite set of r -dimensional integer vectors. The reachability set $R(\mathcal{W})$ of \mathcal{W} is the set

of vectors of the form $d + w_1 + w_2 + \dots + w_s$ such that $\forall i \in 1 \dots s, w_i \in W$ and $d + w_1 + w_2 + \dots + w_i \geq 0$.

Property 7.10 (Coverability) *It is decidable of a VAS \mathcal{W} and a point x whether $R(\mathcal{W})$ contains a point $y \geq x$.*

Definition 7.11 (VASS) *A VASS is the product of an automaton and a VAS.*

The Figure 7.2 describes such a VASS.

7.2.3 Effective checking

The practical checking of the property is now the following. We want to rely on the good properties of VASS, in particular decidability of the coverability property. Orphan freeness being expressed as such a problem, we now face the difficulty of computing a VASS for our CAP system.

Therefore, we rely on our abstract interpretation framework to build such representation. In the next section, we propose an abstract domain to be used in our framework that computes a VASS for each address. It represents in an abstract way both sequences of interfaces and associated mailboxes. Our abstract domain models interfaces by the program point defining them. It merges recursive instances of the same behavior, independently of internal values. This allow us to obtain a finite automaton for each address.

The partitioned abstract domain presented in Chapter 4 solves the dimensional problem, fixing the number of address values by the number of binders and merging VASSs associated to recursive instances of the same binder.

The solving protocol is then the following:

1. we rely on our abstract interpretation framework to build a VASS for each binder;
2. using the resulting abstract element, we compute for each VASS state an over-approximation of possible message labels. This computation relies on VASS decidability properties;
3. we check the property for each message label in each binder VASS automaton: each message label available at each node must be consumed in every possible path considering weak fairness among the different behavior branches of the same behavior set;
4. in order to guaranty the absence of liveness orphans, each node is the VASS automaton must be able to compute a transition: it necessitates an over-approximation of mailboxes at each step.

In the case where only the first three steps can be done, the system is guaranteed to be orphan free considering the safety orphan definition.

In the case where all steps can be done, all messages are guaranteed to be consumed.

7.3 ABSTRACTING MAILBOXES AND INTERFACES

The current section defines an abstract domain that represents mailboxes and sequences of interfaces (behaviors) associated to actors. It has to be used under the partitioned abstract domain presented in Chapter 4. We first give the intuitions behind the domain definitions and its primitives. We then formalize a first version of the domain. A third part addresses some domain enhancements, allowing to obtain a sufficient precision while keeping soundness and simplicity of presentation.

7.3.1 *Intuition*

The aim of the current abstract domain is to carry information to abstract behaviors associated to actors but also to represent mailboxes. Its use under the partitioned analysis allows us to only focus on representing data for a given address. Then, the internal machinery of the partitioned analysis merges recursive instances of the same name binder.

The main idea is to build a graph where nodes represent actors installed on an address. A link between two nodes denotes a transition consuming the first actor and producing the second, both on the same address (we recall that we work under the partitioned abstract domain, focusing on a single address at once). When an actor is consumed without creating another actor on the current address, we use a *dead* node with the same label to denote a configuration without an installed actor.

Such a graph is therefore sufficient to over-approximate sequences of behaviors. All possible traces can be built from initial nodes in the graph.

Labeling transitions with the matched behavior gives us information on consumed and produced messages, but we may miss messages that are produced by other actors on other addresses. Therefore, in order to keep track of additional messages or initial ones, we associate to each node an abstract local mailbox.

Initial abstract element

The initial element is composed of a graph with one single node if there is an actor on the address in the initial configuration and with the empty graph if there is not. Similarly if there are, initially, some messages sent to the address,

the appropriate abstract local mailbox is associated to initial elements of the graph.

Abstract transitions computation

An abstract transition should mimic a concrete one. It might compare interacting threads, allow the transition to be computed, compute value passing, launch continuations and remove interacting threads if necessary. In our current case, one can easily project such protocol on our structure. We first check whether there is enough messages for the transition to be computed. We forget about value passing and update both the graph and local abstract mailboxes depending on the computed continuation.

In practice, we do not constrain the transition. The synchronization step would require to have an abstract representation of all messages available at each node. What we have not. It can be however computed as it will be later in the checking phase of our orphan freeness checking. But the mailbox computation, relying on both abstract local mailboxes and graph transition, is costly. Therefore we choose to rely on other abstract domains such as the control flow one (cf. Section 3.3.2), or the occurrence counting one (cf. Chapter 5). In case of a need of great precision, it is always possible to activate such synchronization. But each abstract transition will then necessitate the mailbox computation and will drastically slow down the analysis.

A drawback of partitioning properties depending on address values is the precision we obtain when launching continuations outside unit. We first give an insight of the inside unit continuation launching then we present the problematic behind the outside unit continuation launching. We recall that the launching in the current interacting unit corresponds to the launching of threads bound to the interacting threads address, in opposition to the launching outside the interacting unit that corresponds to threads launched on other addresses.

LAUNCHING CONTINUATION INSIDE THE PARTITION UNIT When launching continuation in the current active partition unit, we build a new node and add a transition between the current active actor and this target node. If no actor is launched, then the new node is a dead one. Else it is labeled by the program point of the launched actor.

This new edge is labeled by the behavior branch used and the set of messages launched on the address.

LAUNCHING CONTINUATION OUTSIDE THE PARTITION UNIT Threads launching outside the current active unit is much more complex. In the above case, we have the information about the active actor that is consumed by the transition. Therefore we exactly knew who was the last previous actor in the graph that launches the continuation. Launching a message or an actor on another address

requires to over-approximate the best last ancestor. Then for each ancestor, we add an edge as in the preceding case. If no actor is launched but only messages sent, we have to add such messages to the local abstract mailbox of the ancestor successors.

The linearity hypothesis allows us to only consider dead node as valid ancestors when launching actor outside unit (in order to avoid multiple actor on the same address).

OVER-APPROXIMATING ANCESTORS Over-approximating ancestors has to be handle with care. A wide over-approximation would saturate the graph with edges and would not allow to observe anything. The over-approximation presented in the following relies on a sound but loose version: we consider all nodes as potential ancestors. In fact, the identity marker associated to each thread in the non standard encoding is sufficient to identify the exact best ancestor. It describes possible sequences of transition that led to its creation. A second approach, the one we implemented, relies on the control flow approximation of such markers as presented in Section 3.3.2 page 60 in order to restrict the set of potential ancestors.

Example 12 (Linear cell based systems (continued)) *Let us illustrate the continuation launching outside active unit on our previous examples. The Figure 7.3 describes an abstract transition computation in the S_2 system. Dotted edges represent the ancestor over-approximation. In the current unit, the new node $(5, \mathbf{T})^2$ is added and linked by a transition producing no message. In the other partition unit, the message labeled `rep` has to be added. Since we have not so much information yet, it is added to the local mailboxes of each nodes. The abstract local mailboxes have an unbounded number of messages since we rely on a non relational analysis in order to count occurrences of message labels. The use of widening gets the fixed point while widely over-approximating the upper bound of such mailboxes.*

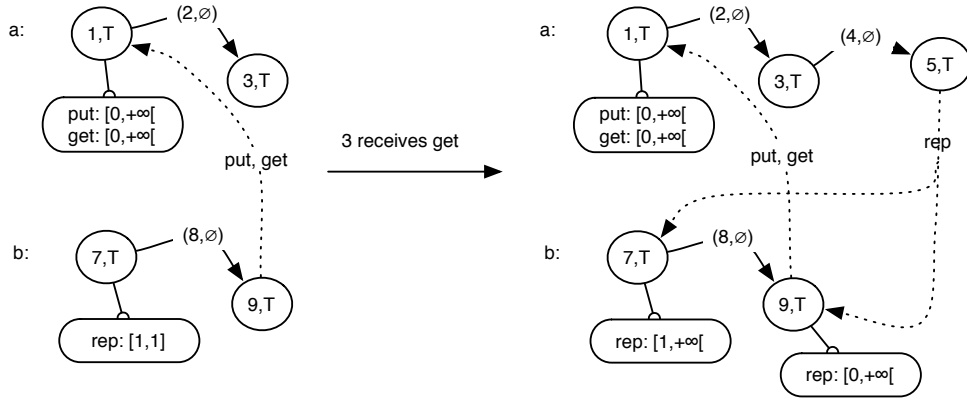
The Figure 7.4 gives the final abstract element obtained in both S_1 and S_2 systems.

7.3.2 Abstract domain

We now formally introduce our new abstract domains allowing to represent the possible sequences of behaviors associated to actors as well as the set of messages available at each step of these sequences. These domains are used under the partitioned abstract domain defined in the Chapter 4. Each partition represents the interfaces and mailboxes of actors associated to the same name binder.

We first define the domains. Then we state the principal proof.

2 We denote by (p, \mathbf{T}) normal nodes and by (p, \mathbf{F}) their associated dead nodes.

Figure 7.3 Message sending over-approximation outside partition unit.

Mailbox lattice

We first introduce a generic abstract domain $\text{MX}^\#$ defined over the lattice $(\text{MX}^\#, \sqsubseteq_{\text{MX}}, \sqcup_{\text{MX}}, \perp_{\text{MX}}, \gamma_{\text{MX}})$ where $\text{MX}^\# = \wp(\mathcal{M}_l \times \mathbb{N}^\#)$ denotes multisets of message labels in \mathcal{M}_l with abstract occurrences in $\mathbb{N}^\#$.

We introduce a pre-order $(\text{MX}^\#, \sqsubseteq_{\text{MX}})$ between abstract multisets.

Concrete mailboxes as multisets of labels are defined as the set MS . Abstract elements of $\text{MX}^\#$ are mapped to MS , families of natural numbers indexed by \mathcal{M}_l by the monotonic concretization γ_{MX} . We recall that \mathcal{M}_l denotes the finite set of message and behavior branch labels.

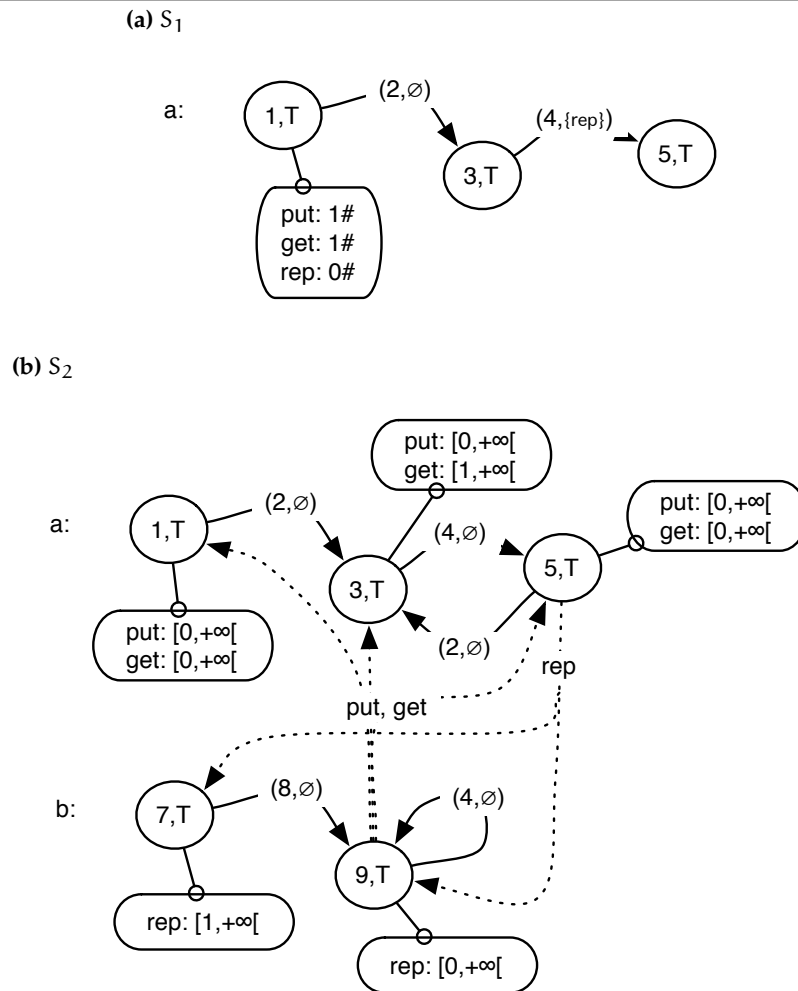
We introduce several usual primitives: a representation \perp_{MX} of the bottom element, an abstract union \sqcup_{MX} , a widening operator ∇_{MX} , an abstract counterpart $+_{\text{MX}}^\#$ to the binary addition and an abstract counterpart $-_{\text{MX}}^\#$ to the binary subtraction. We also require the abstraction of some elementary families: an abstract element 0_{MX} to represent the empty multiset and $\forall m \in \mathcal{M}_l$, an abstract element $1_{\text{MX}}(m)$ to represent the singleton containing the family δ^m which associates 1 to the element m and 0 to any other element. These primitives must satisfy the soundness properties of families of natural numbers indexed by \mathcal{M}_l as defined in Definition 3.7, page 63. This domain is comparable to the domain \mathcal{N}_{γ_c} built on \mathcal{M}_l .

We instantiate this generic domain by the interval domain presented in Section 5.1.1, page 100.

Behavior graph lattice

Then, we define a second lattice $(\text{G}^\#, \sqsubseteq_{\text{G}}, \sqcup_{\text{G}}, \perp_{\text{G}})$ representing the sequences of behaviors. The lattice $\text{G}^\#$, abstracting causality between actors, is a finite directed multigraph built over the set $\text{N}^\# \triangleq \mathcal{L}_p \times \mathbb{B}$ of nodes. A node (p, b)

Figure 7.4 Resulting abstract mailboxes and interfaces for the linear cell-based systems.



could be initial or not and denotes an actor on program point $p \in \mathcal{L}_p$ still in the configuration or consumed, depending on the value of the boolean argument.

Edges denote causality and are labeled by a pair $(p^b, ms) \in (\{\odot\} \cup \mathcal{L}_b) \times MS$ where p^b denotes the matched behavior branch and ms the multiset of launched message labels. The \odot program point is used in edges from **F** nodes to **T** ones. These edges do not represent a transition on the partition unit address and should not carry any behavior branch used to produce the threads since they did not consume any thread on the unit addresses.

We define a pre-order on such graphs using the usual graph inclusion of edges extended using multiset inclusion for edges. We also define the other usual operators. The bottom element \perp_G denotes the empty graph; the abstract union \sqcup_G merges nodes with the same label as well as edges sharing the same source, target and label; the widening operator ∇_G is defined as the abstract union since there is a finite number of both nodes and possible edges. We provide some primitives allowing to build elementary elements: creating a new node $\text{node}((p, b), \text{init})$, initial or not, for a given label $(p, b) \in N^\#$ or adding an edge $\text{edge}(n1, \text{label}, n2)$ between two nodes. The primitives nodes and edges return, respectively, the set of nodes and the set of edges of a graph.

Main domain

The principal abstract domain is parametrized by the abstract domain $MX^\#$. with a function associating abstract multisets to nodes $(\{\epsilon\} \cup N^\#) \rightarrow MX^\#$. The ϵ value allows to define messages available to initial nodes.

$$I^\# \triangleq G^\# \times ((\{\epsilon\} \cup N^\#) \rightarrow MX^\#)$$

The abstract multiset associated to a node denotes its local abstract mailbox. It approximates the set of messages that could be sent to the immediate successor of the actor after its launching and before its consumption.

The pre-order \sqsubseteq_I , the bottom element \perp_I , the join operator \sqcup_I and the widening ∇_I are defined component-wise and using the point-wise extension of operators on $MX^\#$ to functions on $N^\# \rightarrow MX^\#$.

Abstract elements over-approximate both emissions, *i.e.* the different mailboxes available at each step of an actor behavior evolution, and receptions, *i.e.* the future behavior sequences that an actor may assume. The abstract mailbox available at each node can be computed using the interface graph and local abstract mailboxes. We denote by $\text{mailbox}(n, i)$ such mailbox for the node n in the abstract element i . It can be expressed as a least fixed point of the incoming paths of the considered node. It is used to state soundness but its effective computation is defined in the next section.

We relate abstract elements to sets of concrete configurations (u, C) in $\Sigma^* \times \mathcal{C}$ (we recall that \mathcal{C} is the set of all non standard configurations) by the monotonic concretization function γ_I :

Concretization

The monotonic γ_I function relates elements of $I^\#$ to reachable concrete configurations (u, C) where u denotes the sequence of transitions that leads to the configuration C .

We recall that we consider only linear systems, *i.e.* with at most one actor *per* address and that we intend to use this domain under the partitioned abstraction.

$$\left. \begin{array}{c} (u, C) \\ \left\{ \begin{array}{l} \forall a \in \mathcal{L}_v \times \mathcal{M}, \\ \left\{ \begin{array}{l} (p^a, \mathbf{T}) \text{ node of } g \\ \text{mailbox} \in \gamma_{MX}(\text{mailbox}^\#((p^a, \mathbf{T}), (g, mx))) \\ \text{when } \left\{ \begin{array}{l} \exists (p^a, id^a, E^a) \in C_{|a} \wedge p^a \in \mathcal{L}_a \\ \wedge \text{mailbox} = C_{|a} \setminus \{(p^a, id^a, E^a)\} \end{array} \right. \\ \left\{ \begin{array}{l} \text{either } C_{|a} \in \gamma_{MX}(mx(\epsilon)) \\ \text{either } \left\{ \begin{array}{l} \exists p \in \mathcal{L}_p, s. t. (p, \mathbf{F}) \text{ node of } g \\ \text{and } C_{|a} \in \gamma_{MX}(\text{mailbox}^\#((p, \mathbf{F}), (g, mx))) \end{array} \right. \\ \text{otherwise} \end{array} \right. \end{array} \right\} \end{array} \right\} \end{array} \right\} \subseteq \gamma_I((g, mx))$$

Let us describe the different parts of this concretization. As it is used under the partitioned analysis, we focus on each address one by one.

An abstract graph containing a node (p, \mathbf{T}) is able to describe any possible configuration containing an actor on this address with an associated set of messages, its mailbox, contained in the concretization of the abstract mailbox associated to the node (p, \mathbf{T}) .

A configuration that does not contain an actor can also be represented as soon as it exists a dead node (p, \mathbf{F}) which has an associated abstract mailbox that overapproximates the configuration. The case ϵ handles initial abstractions without any actor.

Semantics primitives

We now introduce a set of primitives that allow us to manipulate abstract elements. We introduce the abstract synchronization in order to fully define the domain. But as explained before, we do not rely on this primitive in our implementation. The main definition is the abstract launching. Such primitives must update both abstract mailboxes and the abstract representation of future interfaces.

ABSTRACT SYNCHRONIZATION The synchronization primitive uses the abstract information to allow or forbid an abstract transition. It relies on the mail-

box approximation that is presented in the next section. It necessitates a fixed point computation and is therefore costly.

Definition 7.12 (Abstract synchronisation) Let $(g, mx) \in I^\#$ be a directed multi-graph $g = (\text{nodes}, \text{edges})$ associated to a map mx from nodes to abstract multisets, denoting an abstract interface and its abstract local mailboxes. Let \mathcal{R} be a formal rule. Let $p^a, p^m \in \mathcal{L}_p$ be the program points in (p^k) associated to the interacting actor and message thread, respectively. Let $(parameter^k)$ be the formal variables associated to interacting threads.

Let m be the message label associated to the message program point p^m . We denote by $mx \in MX^\#$ the mailbox defined as

$$mx \triangleq \text{mailbox}((p^a, \mathbf{T})) \text{--}_{MX} \mathbf{1}_{MX}(\{m\}).$$

We define the primitive `sync` as:

$$\text{sync}(\mathcal{R}, (p^k), (parameter^k), (g, mx)) \triangleq \begin{cases} (g, mx) & \text{if } mx \neq \perp_{MX} \\ \perp_I & \text{else} \end{cases}$$

LAUNCHING PRIMITIVES The `ancestor_launch` primitive modifies the abstract element to reflect the continuation launching when computing a transition.

Definition 7.13 (Abstract continuation launching with ancestor) Let $p \in \mathcal{L}_p \cup \{\odot\}$, $p^k \in \mathcal{L}_p$ be program points respectively denoting the ancestor program point and the tuple of interacting threads, $conts$ be the static continuations extracted from the

term and $i \in I^\#$ be an abstract element. We denote by p^α the actor program point of (p^k) . We define the primitive $\text{ancestor_launch}(p, (p^k), \text{conts}, i)$:

$$\begin{aligned} & \text{ancestor_launch}(p, (p^k), \text{conts}, (g, \text{mx})) \triangleq \\ & \text{if } p = p^\alpha \text{ then} \\ & \quad \text{if } \exists (p', E_s) \in \text{conts}, p' \in \mathcal{L}_a \text{ then} \\ & \quad \quad g \sqcup_G \text{edge}((p, T), (p^1, \text{messages}), (p', T)), \text{mx} \\ & \quad \text{else} \\ & \quad \quad g \sqcup_G \text{edge}((p, T), (p^1, \text{messages}), (p, F)), \text{mx} \\ & \text{else if } \exists (p', E_s) \in \text{conts}, p' \in \mathcal{L}_a \text{ then} \\ & \quad \text{if } p = \odot \text{ then} \\ & \quad \quad g \sqcup_G \text{node}((p', T), \text{true}), \text{mx} \\ & \quad \text{else} \\ & \quad \quad g \sqcup_G \text{edge}((p, F), (\odot, \text{messages}), (p', T)), \text{mx} \\ & \text{else} \\ & \quad g, \lambda(p', T). \begin{cases} \text{mx}(p', T) +_{\text{MX}} \sum_{p \in \text{conts}} 1_{\text{MX}}(\text{label}(p)) \\ \quad \text{when } (p', T) \in \text{succ}((p, T), g) \\ \text{mx}(p', T) \quad \text{otherwise} \end{cases} \\ & \quad \text{where } \text{messages} = \{p \mid (p, E_s) \in \text{conts} \wedge p \neq p'\} \end{aligned}$$

The function label denotes the message label associated to a message thread. It is extracted from the associated partial interaction.

The ancestors primitive allows to over-approximate the last actor associated to a name in order to represent causality dependencies.

Definition 7.14 (Abstract ancestors) Let $(\text{Id}^\#)_k$ be a tuple of regular approximation of interacting thread marker trees built over actor program points. Let $i \in I^\#$ be an abstract element. We define the primitive $\text{ancestors}((\text{Id}^\#)_k, i)$, which computes the set of possible ancestor program points, as follows:

$$\text{ancestors}((\text{Id}^\#)_k, (g, \text{mx})) \triangleq \{\odot\} \cup \left(\text{nodes}(g) \cap \left(\bigcup_k \text{pp}(\text{Id}^k) \right) \right)$$

where $\text{pp} : \text{Shape} \rightarrow \wp(\mathcal{L}_p)$ denotes the function that maps regular approximations to the sets of letters contained in the approximated words.

In other words:

- for each k -th interacting threads, we consider the node of g build on program points present in the marker approximations $(\text{Id}^\#)_k$;

- the possible ancestors of the launched threads can be any of the actor program point present in these restrictions as well as the special program point \odot which denotes no dependency.

Finally, the main launch primitives are now defined. The following primitives must be defined in order to fulfill the requirements of Definition 4.7 of the partitioned abstract domain.

Definition 7.15 (Abstract launching) Let $p^k \in \mathcal{L}_p$ be program points denoting interacting threads, $(\text{Id}^\#)_k$ be a tuple of regular approximation of interacting thread marker trees built over actor program points, b be a boolean stating whether we are launching threads in the current unit or not, cont_s be the static continuations extracted from the term and unit_a be an abstract element of $\mathbb{I}^\#$. We denote by p^a the actor program point of (p^k) . We define the launch primitives:

- The `update_interacting` primitive relies on the `ancestor_launch` primitives. It is used in the interacting unit of the partitioned abstraction and represent threads launched in the current active unit. It calls the `ancestor_launch` primitive with the best ancestor: the interacting actor. Each possible continuation is launched and the union of the element obtained computed.

$$\text{update_interacting}(\mathcal{R}, (p^k), (p_i^k), \text{synced}, a, \text{unit}_a, \text{cont}_a) \triangleq$$

$$\bigsqcup_{\text{cont} \in \text{cont}_a} \text{ancestor_launch}(p^a, (p^k), \text{cont}, \text{unit}_a);$$

- The `launch` primitive is defined in a similar manner. In the partitioned abstraction, it is used to launch thread in non interacting unit. In our case, we need to over-approximate the best ancestor.

$$\text{launch}(\mathcal{R}, (p^k), (p_i^k), \text{synced}, b, \text{unit}_b, \text{cont}_b) \triangleq$$

$$\bigsqcup_{p \in \text{ancestors}((\text{Id}^\#), \text{unit}_b)} \left(\bigsqcup_{\text{cont} \in \text{cont}_b} \text{ancestor_launch}(p, (p^k), \text{cont}, \text{unit}_b) \right)$$

$$\text{where } (\text{Id}^\#) = (\top^\#)_k;$$

- Finally the last primitive `launch_beh` represents behavior threads launching. In the concretization, no behavior thread is constrained. We have:

$$\text{launch_beh}(\mathcal{R}, (p^k), (p_i^k), \text{synced}, \text{beh}, \text{cont}_{\text{beh}}) = \text{beh}.$$

Operational semantics

INITIAL ELEMENT

Definition 7.16 (Initial abstract configuration) The initial element is obtained by

- building the graph with one initial node for each actor in the initial term;
- and associating the abstract mailbox built with the initial sent messages associated to the ϵ value (which denotes in this case the initial abstract mailbox).

ABSTRACT TRANSITIONS We use the above primitives to describe the initial abstract element and the abstract transition rule. The initial elements abstraction $C_0^\#$ is computed using the `ancestor_launch` primitive with the threads present in the initial configuration init_s and the \odot ancestor. We do not provide any interacting threads and call it with the argument $(\)$ instead of (p_k) :

$$C_0^\# = \text{ancestor_launch}(\odot, (\), \text{init}_s, \perp_I).$$

The partitioned abstract domain semantics, used to manipulate our abstract mailbox and interface domain, relies on the abstract primitives. We informally recall the different steps of the abstract transition computation:

- find a transition rule and a tuple of interacting threads;
- choose a unit that can be associated to both the actor and the message threads, using the control flow part of the abstract element;
- check that the abstract element associated to this unit satisfies the synchronization constraints defined by both the actor and the message threads;
- launch threads in continuation, when the transition is fireable:
 - compute, using the control flow part, the possible units updated by the launched threads with their subset of the continuation;
 - for each of these units, identify the possible ancestors of the launched threads and update the abstract element.

Soundness

The following theorem states the soundness of our domains.

Theorem 7.17 $(I^\#, \sqsubseteq, \sqcup, \perp, \gamma_I, C_0^\#, \nabla)$ is a sound abstraction when used under the partitioned abstract domain and considering linear systems. It satisfies the sound assumption of Definition 4.7.

Proof 7.18 The proof can be found in A.5.

7.3.3 Improvements preserving soundness

Reduction with control flow information to enhance causality research.

A necessary step in the previous domain operational semantics is the use of information about the identity marker of interacting threads in case of threads launched outside the interacting unit. These marker approximations are used in order to over-approximate the last actor on the address to represent causality.

This information is present in the control flow part of our partitioned abstract element during the abstract computation. A first and immediate solution is to

compute a reduction between both parts of the element during the transition computation using a simple intersection.

This is why we use the enhanced flavor of the partitioned abstraction, as presented in Section 4.4, built on the shape approximation of values and markers. In the following, the new shape value is only used when approximating ancestors. The only primitive which change is then the launch primitive. All other are left as is, with a new argument in the call, the shape value. Our primitives become:

- $\text{update_interacting}(\mathcal{R}, (p^k), (pi^k), \text{synced}, a, \text{unit}_a, \text{cont}_a, \text{shape}) \triangleq$

$$\bigsqcup_{\text{cont} \in \text{cont}_a} \text{ancestor_launch}(p^a, (p^k), \text{cont}, \text{unit}_a);$$
- $\text{launch}(\mathcal{R}, (p^k), (pi^k), \text{synced}, b, \text{unit}_b, \text{cont}_b, \text{shape}) \triangleq$

$$\bigsqcup_{p \in \text{ancestors}((\text{Id}^\#)_k, \text{unit}_b)} \left(\bigsqcup_{\text{cont} \in \text{cont}_b} \text{ancestor_launch}(p, (p^k), \text{cont}, \text{unit}_b) \right)$$

 where $(\text{Id}^\#)_k = (\text{shape}(p^k)(I))_k$ ³;
- $\text{launch_beh}(\mathcal{R}, (p^k), (pi^k), \text{synced}, \text{beh}, \text{cont}_{\text{beh}}, \text{shape}) = \text{beh}.$

A second more accurate approach to enhance marker approximation, which has not yet been implemented in our analyzer, is to compute a partitioned marker tree regular approximation in order to separate the markers associated to different addresses but on the same actor program point. This solution seems more interesting for the causality approximation as it would allow to consider only the root of such marker tree approximation denoting the last transition on the current address.

In the current analysis, using a non partitioned control flow, the transitions involving actors on a given address are scattered among markers.

Reducing occurrences in mailboxes using a relational abstraction.

Finally, an efficient yet simple improvement is to constrain occurrences in multisets of mailboxes by a relational abstract domain such as linear equalities between occurrences of message labels and computed transitions. Occurrences in multisets are then replaced by a product of abstract domains, a non relational one with a relational one, both of them under a reduction as presented in Chapter 5.

³ We recall that the variable I denotes the identity marker of threads in the control flow approximations, in particular in this regular approximation of thread environment shape.

7.4 ENSURING ORPHAN-FREENESS

Once the abstract interpretation machinery is launched, fed with the appropriate domains, we obtain an abstract element least fixed point of the abstract computation that over-approximates set of reachable non standard configurations (u, C) in $\Sigma^* \times \mathcal{C}$.

We now rely on such an abstract element, denoting properties satisfied by the collecting semantics, in order to ensure orphan freeness. The above abstract domain, used under the partitioned abstract domain, yields for each actor name α , an over-approximation of sequences of behaviors associated to it as well as its mailboxes evolutions, as a VASS.

Initially we defined orphans and orphan freeness as a temporal property expressed on configurations using the notion of mailboxes and sets of future interfaces. Then we proposed an expression as a coverability test over VASS. We now face a difficulty: how the over-approximation of terms could soundly approximate the notion of future interfaces associated to addresses ?

A first point is the mailbox representation. Our abstract element does not provide an abstract mailbox associated to each node, but rather a local view of it. Computing these mailboxes could necessitate another step of abstraction, in order to preserve efficiency.

A second point considers the construction of a sound under-approximation of the future interfaces available, considering the graph computed.

We now opine on these different points. We first address the mailbox computation. A second part considers the different sources of over-approximation and discusses the quality of the resulting element depending on the cause of approximations. Then we give the algorithm ensuring the property. Finally we present the actor stuck freeness checking.

7.4.1 *On effective mailbox computation*

When we need to check orphan freeness, we have to compare mailboxes with future interfaces. We have to check for each state that a reachable state involves a computation consuming present messages. Therefore, we need a characterization of what the mailbox may contain at each step. The results presented in [67] give us that the reachability set is not always computer representable. Therefore we cannot find a concrete mailbox representation for each node.

Due to the inherent abstraction of our abstract interpretation framework, we may have already merged different states, either actors on different recursive addresses or actors on recursive instances of the same definition, *i.e.* on the same program point. The approximated mailbox that we compute for a given graph node represents the mailbox associated to any actor the node abstracts.

We now present two different approaches to compute such abstract mailbox at each node. The first one relies on a fixed point computation using widen-

ing to converge in a finite time. The second one which seems more precise but also more costly involves a graph search algorithm and relies on HIGMAN lemma [61] to widen growing mailboxes in an exact manner though.

Explicit abstract computation as a fixed point

A first approach expresses the mailbox associated to a node as a fixed point. A node mailbox is defined as the union of the node mailboxes targeting it, considering consumed message and produced ones, augmented with the abstract local mailbox.

Definition 7.19 (Mailbox computation step) *Let n, n' be two nodes of an abstract element $i \in I^\#$ and $e \in i$ be an edge from n to n' in i . Let mx and $local_mx$ be two functions mapping nodes of i to element of $MX^\#$. We define the primitive $MxUpdate$:*

$$MxUpdate(e, mx, local_mx) \triangleq \\ (mx(n') -_{MX}^\# \mathbb{1}_{MX}^\#(\{m\})) +_{MX}^\# local_mx(n) +_{MX}^\# \mathbb{1}_{MX}^\#(l)$$

where e is labeled by the behavior branch of label m launching the multiset of message labels l .

Definition 7.20 (Fixed point definition) *Mailboxes of an abstract element $((nodes, edges), local_mx) \in I^\# \times ((\mathcal{L}_p \times \mathbb{B}) \rightarrow MX)$ are defined as the least fixed point of the following \sqcup -complete endomorphism \mathbb{M} over the point wise extension of abstract mailboxes over graph nodes $((\mathcal{L} \times \mathbb{B}) \rightarrow MX^\#)$:*

$$\mathbb{M}(mx) \triangleq (\lambda x. local_mx(x)) \\ \sqcup_{MX} \left\{ mx' \text{ s.t. } mx'(n) = mx(n) \sqcup_{MX} \bigsqcup_{e \in edges} MxUpdate(e, mx, local_mx') \right\}$$

where $local_mx'$ is defined as the extension of $local_mx$ where initial nodes are associated to the abstract mailbox $local_mx(\epsilon)$ and \sqcup_{MX} the point-wise extension of MX to maps in $N^\# \rightarrow MX$.

$$mailbox^\# = lfp_\perp \mathbb{M}$$

The use of a widening operator to ensure convergence in a reasonable finite number of iterations gives an over-approximation of messages available at each step of the actor life.

Testing message occurrence using HIGMAN lemma

Another approach relies on a graph search algorithm, for example a depth-first search. Initial nodes are associated to the initial local abstract mailbox. When

reaching a visited node in the depth-first search, we have to compute the effect of the cyclic paths in the VASS on the node mailbox. The HIGMAN lemma states that the property of a binary relation S to be unavoidable is also a property of the binary relation S^* . In our VASS, with as a binary relation S the usual order \leq of natural numbers extended point-wise to natural vectors, we obtain that there exists a reachable state for the node with a bigger mailbox than the current one.

Let us first give some definitions and then explain their use.

Definition 7.21 (Unavoidable binary relation) *The relation S is unavoidable if for all infinite sequence $\alpha_0, \dots, \alpha_n \dots$, there are two elements α_i and α_j such that α_i precedes α_j and α_i is S -related to α_j .*

The relation \leq on natural numbers is unavoidable.

Lemma 7.22 (HIGMAN's lemma) *Given a binary relation S , if S is unavoidable then S^* is unavoidable as well.*

In practice, when detecting cyclic paths while searching the VASS, we may obtain another vector associating different abstract occurrences to the different message labels. The HIGMAN lemma allow us to widen to infinity the occurrences of message labels that would grow during a computation of the cyclic path. This gives us a sound over-approximation of mailboxes while iterating only once for each cyclic path. This approximation, very coarse, is only used to test for the possible occurrence of a message label. It is not intended for numerical purposes, for instance.

7.4.2 Where over-approximations comes from

In the above presentation, approximations are numerous, the control flow abstract domain over-approximates interactions between threads and may allow transitions that will not occur in practice, especially in our high order calculus, where values carry behaviors; the occurrence counting abstract domain keeps track of the threads presence but may also allow transitions with threads already consumed; finally the partitioned abstract domain merges recursive instances of the same name binder and the outside active partition unit launching may cause a great loss of precision when abstracting causality. Last but not the least, the effective mailbox representation at each node in the graph necessitates another step of approximation.

All these sources of approximation allows us to reach the property computation in an acceptable time and to bridge the gap of undecidability for certain kinds of properties. These approximations are also sound in the sense of the abstract interpretation. Concretizing resulting abstract element does not *forget* possible reachable CAP non standard states.

However one can reasonably ask whether the orphan freeness checking is still meaningful in presence of such abstractions. Checking the property basically requires a representation of mailboxes and of future interfaces and then checking that the first is included in the second. A quick look at this, gives us that if we over-approximate the first (mailboxes) and under-approximate the second (future interface) then it is sound to check the property. But in our case, we over-approximate both.

The over-approximation of mailboxes is not a problem. The more wide the abstraction is, the more difficult it is to ensure orphan freeness. On the other hand, the over-approximation of future interfaces may look like an unsound idea. Spurious edges would allow to consumed messages and therefore would allow to give a false (and unsound) answer.

Let us now enumerate all sources of abstraction and relate them to the orphan freeness checking. We consider the different kinds of node and their edges. We recall that a node (p, \mathbf{T}) denotes an actor installed on the current address, it can either denote a static installation (*i.e.* with a syntactic behavior set) or a dynamic one (*i.e.* depending on a variable value).

- In the first case, there is no possible spurious edge. If such node is accessible then it has the capacity to handle any message label of its behavior set. When checking such node, all paths from this node have to be checked, considering weak fairness among them in case of infinite paths in the graph.
- In the second case, our abstraction guarantees us, if the node is accessible, that an actor is installed with a behavior. All edges from this node can be partitioned in their different defining behavior sets, *i.e.* the different values defining its behavior. A control flow over-approximation may have introduced a spurious behavior value but at least one of them is true. The node has to be checked as before but considering each behavior set one by one. A spurious behavior value may break the property. But if all values satisfy it, we did not miss any orphan.
- Finally, let us consider dead nodes, nodes (p, \mathbf{F}) . Nothing guarantees us, by construction, that the successors of such nodes are not spurious, since no actor is launched. Furthermore, edges from dead nodes are added from other units and are subject to wide over-approximations (see Section 7.3.2). When checking such nodes, they have to be identified as terminal nodes when building maximal traces. In practice, the use of dead nodes, *i.e.* the re-installation of an actor after its death, would often fail the property checking.

7.4.3 Checking orphan-freeness: under-approximating interfaces

Once the fixed point of the abstract collecting semantics computed, the mailboxes over-approximated at each node, relying on either the fixed point expression or the HIGMAN lemma-based approach, we have to check that, at each node, the mailbox is included in the future interfaces. We now give the algorithm that performs such a check. It relies on the above remark concerning soundness of the interface over-approximation.

Definition 7.23 (Checking orphan-freeness) *For each installation program point (p, \mathbf{T}) of a given partition unit and for each message label m present in its associated mailbox:*

- *we check for the existence of some execution path without dead nodes from (p, b) leading to a m branch, in the set of over-approximated configurations, ensuring orphan-freeness;*
- *if such a path contains a node denoting a dynamic installation with some associated behavior values, we check the existence of such paths, still without dead nodes, for each of these behavior values;*

For each node of a given partition unit denoting an actor free configuration (dead nodes), we check that the associated mailbox is empty.

Property 7.24 (Monotonicity) *Given an abstraction of reachable configurations and future interfaces, and following the description above, if this abstraction is declared orphan-free, then any real configuration it represents is also orphan-free.*

Proof 7.25 *The two possible approximations are in dynamic behavior nodes and dead ones. In the first, considering every possible dynamic behavior (even spurious ones) over constrains the set of configurations.*

In the second, systematically considering dead nodes as terminal ones allows to soundly deal with spurious edges added by launching actors outside the current unit.

7.4.4 Checking non stuck actors

A last step to guarantee message consumption is to ensure that actors with non empty mailboxes do not remain stuck. It needs an explicit representation of the mailboxes at each node, using the least fixed point characterization. Each node denoting a static actor must contain at least one message label of its associated behavior.

Definition 7.26 (Non stuck static actor) *Let $i \in \mathbb{I}^\#$ be an abstract element. We denote by $B : \mathbb{N}^\# \in \wp(\mathcal{M}_i)$ the function that maps nodes to the set of behavior branch labels of their immediate successors.*

We have:

$$\forall n, \forall mx \in \gamma_{MX}(\text{mailbox}(n, i)), \exists m \in mx, m \in B(n)$$

Concerning dynamic actors, a similar reasoning as for orphan freeness checking, applies: each behavior set must be checked one by one.

7.5 EXAMPLE ANALYSIS

7.5.1 Example

We now illustrate orphan freeness checking on a simple CAP example. The following example illustrates both replication and behavior passing mechanisms in CAP.

Example 13 (A behavior passing example) *The Figure 7.2 gives the CAP term. The ν operator defines a single address a . Initially one actor (\triangleright) is associated to this address ($[..]$) and four messages (\triangleleft) are sent to it. Two of them contain values denoting behaviors. At this point the actor on a can handle beh messages. After a first transition, there is an actor on address a associated to the behavior carried by the matched beh message. In both cases, the three remaining messages are handled one by one. Depending on the first beh message used, there is or not a final actor on a .*

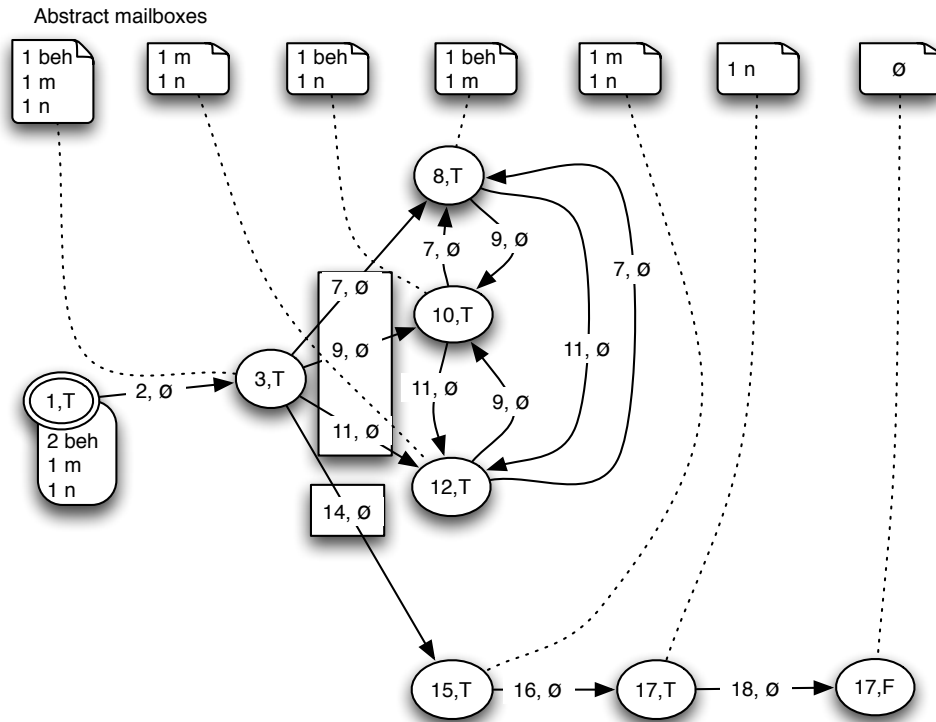
Example 7.2 A behavior passing example.

$$\begin{aligned} & \nu a^\alpha, \quad a \triangleright^1 [\text{beh}^2(x) = \zeta(e, s)(e \triangleright^3 x)] \parallel a \triangleleft^4 m() \parallel a \triangleleft^5 n() \\ & \parallel a \triangleleft^6 \text{beh} \left(\left[\begin{array}{l} m^7() = \zeta(e, s)(e \triangleright^8 s); \\ n^9() = \zeta(e, s)(e \triangleright^{10} s); \\ \text{beh}^{11}(x) = \zeta(e, s)(e \triangleright^{12} s) \end{array} \right] \right) \\ & \parallel a \triangleleft^{13} \text{beh}([n^{14}() = \zeta(e, s)(e \triangleright^{15} [\\ & \quad m^{16}() = \zeta(e', s')(e' \triangleright^{17} [\text{beh}^{18}(x) = 0]) \\ & \quad])]) \end{aligned}$$

7.5.2 Resulting abstract properties

The Figure 7.5 gives the abstract element computed using our abstract domain. We can notice that the only non empty abstract local mailboxes is the one attached to the initial element. Another specificity of this simple example is that no message is launched during transitions.

Figure 7.5 Behavior passing example: resulting abstract element.



7.5.3 Computing mailboxes

The Figure 7.5 also gives the abstract mailboxes computed using the definitions of Section 7.4.1.

7.5.4 Checking orphan-freeness

We can now verify that the mailboxes associated to each node do not contain orphan messages.

The last three mailboxes presented on Figure 7.5 can be easily checked. There is only one path with statically installed actors. The last one is a dead node associated to an empty abstract mailbox.

Nodes $(8, T)$, $(10, T)$ and $(12, T)$ denote actors with a dynamic behavior. However the edges from these nodes denote behavior branches of the same single behavior definition 6. The checking is then similar to the one of statically installed actors.

The node $(3, T)$ denotes a dynamically installed actor. Its edges denoting behavior branches 7, 9, 11 and 14 can be partitioned in two syntactically defined

behavior sets, respectively in program points 6 and 13. One needs to verify that each message label, among beh , m and n , can be received in both behavior sets, which can be easily done.

Finally, the mailbox associated to the initial node, *i.e.* its local abstract mailbox, follows the same algorithm. The beh message label is immediately consumable. The checking for the m and n message labels must follow the above rule, checking among the two behavior sets defined for the dynamic actor on program point 3.

Concerning stuck actor, all abstract mailboxes contain strictly positive occurrences of message labels and are therefore non stuck. The only node associated to an abstract mailbox, that does not satisfy this property, is the node $(17, F)$, but it does not describe an installed actor.

7.6 RELATED WORKS AND DISCUSSION

Few analyses are devoted to the checking of such kind of high level properties in such a context. We first present the previous analyses for orphan freeness developed on CAP and based on type inference. Then we compare our work to the behavioral type systems for π calculus, especially the works by KOBAYASHI. Finally we mention works addressing the encoding of properties into VASS-like structures.

7.6.1 Analyzing CAP by type inference

Preliminary works on orphan freeness checking based on type inference were defined in [25, 27, 28, 29, 31, 41]. Based on $\text{HM}(X)$ type systems, their typing rules produce constraints denoting both causality between sequences of interfaces and message occurrences counting. Finding a solution for the accumulated set of constraints guarantees the property. They face multiple difficulties and are defined on a rather restrictive subset of CAP.

A first one is the high-order flavor of CAP. These type systems forbid sending behaviors in messages.

A second one is the inherent difficulty to compute occurrence counting in type systems. The use of abstract domain is of great use in this case.

A third one is the actor installation on an arbitrary name. In these approaches, terms are also constrained. For example, one cannot write the following term.

$$a \triangleright [m() = \zeta(e, s)(b \triangleright s)]$$

However the last constraint has to be relativized. Our general abstract interpretation framework does not constrain terms in such a way, but considering the particular orphan freeness property checking, our approach does not give accurate results. In fact, when one allows such kind of terms, it necessitates that a

previous actor on b died in order to keep the linearity hypothesis valid. And dead nodes gives often bad results when checking orphan freeness. They are considered as terminal ones, as we cannot ascertain the existence of successor nodes, due to over-approximation of partitioned analysis. Obviously, terminal nodes can handle naturally no message at all.

In [31], the authors rely on complex techniques in order to have a precise representation of interfaces and mailboxes. They use a CTL-based logic to model interfaces and Presburger formulas for mailbox content. Model checking of their formula with respect to their set of typing constraints is then shown decidable. However the complexity of the presburger formula induced by the term typing may be exponential.

Finally all these works consider only safety orphans and their precision did not allow to handle liveness ones, as it requires to represent the minimal occurrence of message labels in configurations.

7.6.2 Behavioral types for the π -calculus

Another very powerful use of type systems for concurrent process is the work by KOBAYASHI [64, 72, 73, 74]. The authors of these works define behavioral type systems for the π -calculus. They mainly associate CCS-like processes to names in order to represent the behavior (emission, reception) of such. They also embedded obligations and capabilities associated to each CCS sub-term in order to represent inter-channel dependencies. Constraining such obligation values can then be used to force a process to compute a certain communication after a number of computed transition. This leads to a lock-freedom type system.

Nevertheless most of the type systems proposed do not allow recursion and were only type checking based for lock-free checking which is the more related to our orphan freeness. In the last work [74], the authors proposed a type inference-based system for deadlock-free processes. In this inference framework, their constraints are less expressive but allow to check without any type annotations. Another problem is that channel creation order is syntactically derived from the term and this could fail checking the property.

7.6.3 Encoding properties into VASS-like structures

A last kind of similar analyses considers the verification of properties by encoding into VASS-like structures, Petri nets for example. They more or less ignore implantation issues and most of them suffer anyway from the state explosion inherent to concurrency.

The work by BALL [14] addresses the verification of multi-threaded software. He model-checks the specification and uses a Karp-Miller covering tree algo-

rithm directly on the model. Without any abstraction, the complexity class of such an analysis is EXPSPACE. The more theoretical work by [58] considers the temporal property verification of concurrent systems communicating through CCS actions. It also relies on a Karp-Miller covering tree algorithm applied on a VASS.

In [9], the authors consider some class of properties for fragments of the asynchronous π calculus and show their decidability by reduction to Petri nets with transfer arcs. A similar approach is presented in [21] by DELZANNO, with decidability results, but it relies on a semi-decision procedure. It is applied to a generic class of process calculi: nominal calculi.

7.6.4 Discussion

We have presented here a method to detect orphan messages in CAP through an abstract representation of mailboxes and future interfaces, as well as abstract domains to over-approximate these properties. Our proposal is able to deal with the full CAP language without any restriction, in particular it handles well the behavior passing. Our proposed analysis takes benefit of all other abstract domains used during the least fixed point computation in order to get accurate abstraction of actor behaviors. Nowadays there was no response to this problem in such a context.

Three drawbacks of the proposed approach can be shown.

- The first one is the bad results obtained when installing an actor outside of the active address. The analysis is able to deal with this case but the results are coarse in case of complex systems. This is easily understandable since one cannot guarantee that the future actor is effectively launched and handles messages. Nevertheless, such a kind of programming habit is often forbidden in practice, only resource owners have usually the right to allocate them.
- Another current weakness is the cost of the second step: the checking phase inside the abstract VASS. As mentioned in related works, the coverability tree algorithm is EXPSPACE in the size of the VASS. The current approach divides the size of VASS drastically. It builds one VASS *per* name binder and merges VASS of the same recursive address instances. This gives us a relative small abstract element for each address and therefore eases its checking.
- Finally the considered properties seem too strong. The finite firing time property could be removed. The checking algorithm has then to be extended in order to ensure infinite paths among behaviors to consume all message labels.

8.1 PACSA: A PRIMITIVE ACTOR CALCULUS STATIC ANALYZER

We implemented all the presented analyses in the PACSA analyzer using the OCAML language.

The general schema of our analysis is presented in Figure 8.1. A CAP term analyzed with PACSA, is firstly parsed and translated into its non standard form. Then the least fixed point of the abstract collecting semantics is computed using a combination of the abstract domains presented in this thesis.

The PACSA analyzer is about 15 kloc and relies on the OCamlGraph library. The OCaml code can be split in three parts:

- A generic part defining basic types and constructions and allowing to compute the least fixed point of the abstract semantics (1,7 kloc).
- A set of CAP specific files defining the lexer and parser of the CAP term but also its encoding into the non standard form: formal rules, partial interactions, automatic program point assignation and extraction function. It also defines functions that rely on the computed abstract element to express CAP specific properties. An encoding of another calculus should require a similar size of code (1,3 kloc).
- A set of abstract domains defining the abstract semantics of non standard terms. They are described later (11 kloc).

Figure 8.1 Overview of our analysis process.

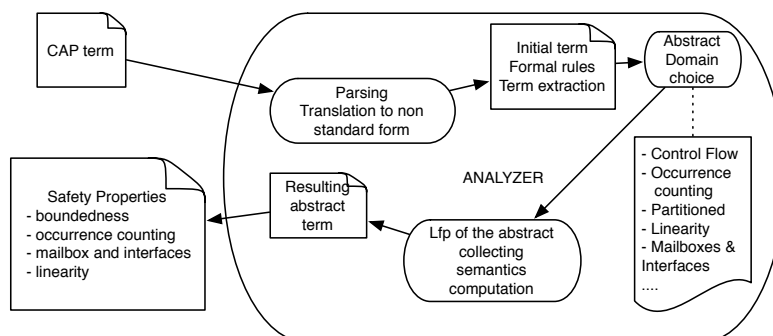


Figure 8.2 Main abstract domain module type.

```

module type DOM =
sig
  type t
  type t_elem
  val bot: unit -> t
  val union: t -> t -> t
  val widen: t -> t -> t
  val leq: t -> t -> bool
  val iter_compute: (rule_t * (Label.t * pa) list) -> t ->
    t_elem * bool
  val iter_launch: rule_t -> Label.t list -> static_conts ->
    t_elem -> t -> t
  val init: static_conts -> t -> t
end

```

8.1.1 *Wide use of Caml modules and functors*

CAML FUNCTORS The methodology of abstract interpretation from its theoretical formalization to its effective implementation really fits a functional approach. Therefore the use of the module and functor features of ML languages gives an elegant and sound approach to the development of abstract domains.

A set of module types are defined for each kind of abstract domain: from the more general one, as presented in Figure 8.2, to the specific one like control flow, as in Figure 8.3, or numerical abstraction, as in Figure 8.4.

The use of functors, building modules from argument modules, allows to design complex abstractions in a very generic manner while preserving type information. For example, we defined the Cartesian product functor or the reduction functor building new sound domains in terms of abstract interpretation in an easy manner.

The genericity available with functors allows us to define a prototype which is as much as possible independent of the analyzed calculus. Later extensions could be to handle calculi with an existing non standard form like π , Ambients, spi, etc, or to define a new calculus inside the framework, while already having a set of abstract domains available. The Figure 8.5 describes the module type of a calculus. Any implementation of a new calculus must define such a module.

Therefore the prototype PACSA was developed while keeping in mind this notion of genericity. It gives us approximately one hundred of functors and modules definitions.

HIERARCHY OF MODULES AND FUNCTORS The most advanced abstraction used is built using all the preceding mentioned abstract domains. It is mainly a reduced product of an occurrence counting domain and a linearity abstraction

Figure 8.3 Control flow abstract domain module types: atoms and molecules.

```

module type ATOM_V =
  sig
    type t
    val leq: t -> t -> bool
    val bot: Label.t -> t
    val epsilon: Label.t -> t
    val restrict: VariableSet.elt -> Label.t -> t -> t
    val gc: VariableSet.t -> t -> t
    val get_V: t -> VariableSet.t
    val union: t -> t -> t
    val is_bottom: t -> bool
  end

module type MOL =
  sig
    module A : ATOM_V
    type t
    type atom = A.t
    val inject: atom -> t
    val concat: t -> t -> t
    val proj: int -> t -> atom
    val newT: VarMolSet.t -> t -> t
    val fetch: Label.t list -> t -> t
    val is_bottom: t -> bool
    val sync: (constraint_param * int * constraint_param * int
              * bool) list -> LabelSet.elt list -> t -> t
  end

```

The abstract domain `atom` is used to associate an abstract representation of each thread environment. It gives for each program point a value of type `t` denoting the properties verified by all threads on this program point.

The molecule abstraction is only used to compute atom interactions. It is never used as a final element and therefore does not necessitate the union and leq primitive definitions.

Figure 8.4 Numerical abstract domain module type for the occurrence counting abstraction.

```

module type NUM_DOM = sig
  type t
  type ilabel
  val leq: t -> t -> bool
  val bot: unit -> t
  val union: t -> t -> t
  val inter: t -> t -> t
  val widen: t -> t -> t
  val plus: t -> t -> t
  val minus: t -> t -> t
  val sync: ilabel list -> t -> t
  val zero: unit -> t
  val one: ilabel -> t
  val is_bottom: t -> bool
  val to_string: t -> string
end

```

Figure 8.5 Generic module type for defining a calculus.

```

type pi_type = Replication | Computation

module type PI = sig
  type t
  val get_type: t -> pi_type
end

module type CALCULUS = sig
  module Pi : PI
  type config
  module Label : LABEL with type pi = Pi.t
  module LabelSet : Set.S with type elt = Label.t
  type pa
  type rule_t
  type transition
  val rules : rule_t list ref
  val get_compatibility : rule_t ->
    ((constraint_param * int) * (constraint_param * int)) list
  val get_vpassing : rule_t ->
    ((constraint_param * int) * (constraint_param * int)) list
  val get_components : rule_t -> Pi.t list
end

```

with a partitioned abstraction built over an occurrence counting domain and our interface abstraction. The partitioned domain is also parametrized by a control flow approximation built using the domains presented in Chapter 3. The essential control flow approximation that drives the abstract transitions is used inside this partitioned abstraction.

The overall construction is presented in Figure 8.6.

8.1.2 *Implementation choices*

In the PACSA development we made some implementation choices. We just mention two of them here.

GENERIC APPROACH TO LFP COMPUTATION A first difficulty was to define a fixed point algorithm in a generic way. Generic means here that we should be able to handle more calculi in the future and do not want to redesign this algorithm.

Another thing, more CAP specific, was to soundly deal with message labels. In CAP standard semantics, message labels must coincide when communicating. The non standard encoding of CAP introduced them in environments. However they are syntactically known and we do not to differentiate their recursive instances.

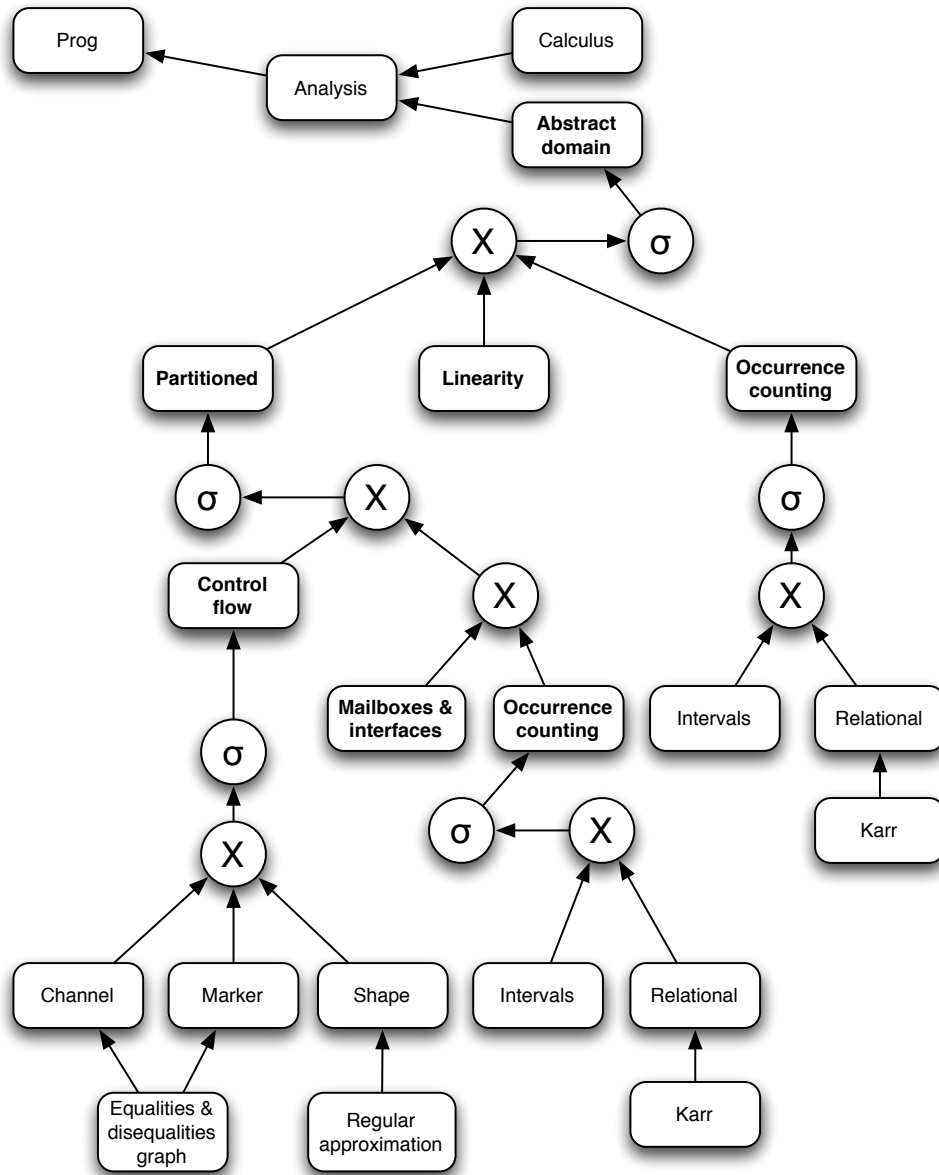
We choose then to build at parsing time a list of possible interactions. This list is based on the set of formal rules defined by the non standard encoding as well as the type of partial interactions exhibited by program points. A first list is then built using, for example in CAP, pairs of statically defined actor program point with message program point and triples of behavior branch program point, dynamic actor definition program point and message program point. In the CAP case, we refine this list by removing pairs or triples involving behavior branch (or static actor partial interactions) associated to different message labels.

When computing fixed points, we update our work list considering updated program points and therefore the set of transitions in which they (could) occur.

Another concern in this fixed point computation is to deal with our backward flow in the linearity abstract domain. In the original framework, the accessible program points at the top-level of the analyzed term are launched once: they cannot be sub-terms of a higher level thread. But our backward flow imposes to update their value. Therefore we add to our list of possible transitions the initial launching. Each time we modify the abstract element associated to an initial program point, we have to re-launch the initial element using the current abstraction.

DEALING WITH REDUCTIONS The attentive reader may have remarked that in the module type of top-level domains, the abstract transition is not specified

Figure 8.6 Hierarchy of modules and functors.



X nodes denote Cartesian products and ρ ones denote reductions.

as a unique primitive but with two. A first one denotes the synchronization step when the second one is the launching step.

This split of abstract transitions in two primitives has two motivations. The first one is to reduce useless computations: when considering costly domains or complex combinations of domains, we do not want to compute the abstract transition in all underlying domains and throw away the results because one domain forbids the transition computation. Separating abstract transitions in two steps allows to rely on intermediate results to effectively launch threads in all abstract domains when needed.

Another use of this separation is to ease reductions. In order to keep the domain definition simple, we provide reduction functions that are available between synchronization and continuation launching.

8.1.3 Domains

We now detail the different files involved in the abstract domain definitions. We briefly mention their content and size. We first present the control flow related abstractions. Then we introduce the occurrence counting related ones. A third part addresses the CAP specific domains. Finally we give an insight on more global abstractions or generic ones.

CONTROL FLOW DOMAINS

- **domains/eq_diseq_graph_domain.ml [1.5 kloc]:** This file specifies a module describing the lattice of equalities and inequalities on a given set of variables V . It uses a graph representation, based on the OCamlGraph library to represent these relations. Nodes are associated to subsets of V denoting equality classes. Edges between nodes denote disequalities between associated equal variables. Union computation splits incompatible node equality classes and remove incompatible edges.
- **domains/channel_graph.ml [1.2 kloc]:** This channel domain implements the control flow underlying abstraction, *i.e.* an atom and a molecule domain allowing to abstract equalities and inequalities among $(p, m) \in \mathcal{L} \times \mathcal{M}$ values between each thread environment variables. It relies on the graph representation introduced in the preceding module built over the set of variables $I(p)$.
- **domains/marker_graph.ml [1.2 kloc]:** Similarly the marker domain provides atom and molecule abstractions with their associated sound primitives to represent equalities and inequalities among marker identities in a thread. It is build over the set $I(p) \cup \{I\}$ where I denotes the identity marker of the abstracted threads.

- **domains/automata.ml [.3 kloc]:** This file defines the regular approximation Reg_L as presented in Chapter 3. It specifies basic constructions to build these simple automata. It also gives the join operator computing the union of two languages.
- **domains/shape_domain.ml [.5 kloc]:** The shape domain is atom and molecule abstractions based on the regular approximation Reg_L . It associates to each variable v of a thread environment abstraction a regular expression of the word $p.id$ where (p, id) is its value. The extra variable I is associated to the abstraction of its thread history marker.
- **domains/control_flow_domain.ml [.2 kloc]:** This file specifies the principal domain for approximating control flow. It relies on an abstract representation of environments as a pair of domains atom and molecule. It then computes the abstract semantics as presented in Chapter 3.

OCCURRENCE COUNTING DOMAINS

- **domains/interval.ml [.3 kloc]:** This file implements the famous interval abstract domain. However we recall that we consider here only positive intervals, for example in the `leq` or `union` functions. Negative ones are only valid when computing interval arithmetic.
- **domains/karr.ml [2.2 kloc]:** This domain is the bigger one. It gives an implementation of the union operator on affine sub-spaces as presented by KARR in [68]. This domain is based on a sparse matrices representation using lists of non empty elements. Column manipulations are eased by a representation of a matrix as a list of sparse blocks. Extracting a column from a block to build a new one is then facilitated in the computation of the different algorithms presented in Chapter 5. The bigger part of this module addresses the traditional triangulation by Gaussian elimination in this sparse structure as well as the different steps of KARR's algorithm.
- **domains/linear_eq_domain.ml [.3 kloc]:** The linear equalities domain is then built upon the KARR's domain implementation. It provides the primitives and operators needed for the main occurrence counting abstraction, such as arithmetic operators or a synchronization primitive.
- **domains/numerical_reduction.ml [.6 kloc]:** The numerical reduction module gives a reduction between intervals and linear equalities built on KARR's domain. It implements the following steps: reducing infinite intervals into finite ones and narrowing finite intervals. It contains the proposed enhancement allowing to reduce more infinite intervals.
- **domains/num_transition.ml [.2 kloc]:** This file instantiates the extension proposed in Chapter 5 providing a more precise synchronization step.

- **domains/occurrence_domain.ml [1.1 kloc]:** Finally the main occurrence counting domain is defined here. It is parametrized by a numerical abstract domain and provides the necessary primitives.

CAP SPECIFIC DOMAINS The following domains are CAP specific. The internal representation relies on the fact that the domains are used only for CAP. Therefore they are less generic than the other ones and are relatively simpler to code or read.

- **domains/linearity.ml [1.3 kloc]:** The linearity abstract domain is given here. It is based on a backward flow and associates a usage mode to variables and binders.
- **domains/mailboxes.ml [1 kloc]:** This module defines the abstract domain of mailboxes and interfaces used in the orphan-freeness property checking. It is quite complex since it necessitates to be used under the partitioned abstraction and has specific functions for launching threads in or outside the current interacting partition unit.
- **domains/partitioned_domain.ml [1.6 kloc]:** The last CAP specific module is the one that implements the partitioned abstract domain. It is parametrized by a control flow abstraction and an abstract domain to be used under the partitioning.

MAIN ABSTRACTIONS

- **domains/abstract_domains.ml [2 kloc]:** This file is very generic and provides multiple domain constructors such as Cartesian product or reduction for top-level domains (occurrence counting, control flow or partitioned) or for low-level ones, e. g. Cartesian product of numerical abstractions.
- **domains/domains.ml [2 kloc]:** This file builds several main abstractions using all the preceding available domains. These domains are then called from the main program depending on the tool arguments.

8.2 RESULTS

The Chapter B gives the result of the PACSA analyzer applied on a complex system. The term analyzed is an extension of the replicating server presented in Figure 2.9. It is composed of four actors:

- the server actor; it answers to messages labeled m and is able to send its behavior and address to the duplicating server;

- the duplicating server; it takes an address and a behavior and replicates the behavior on two new addresses; it binds a behavior that forwards messages to replicated servers and creates a new actor to merge their reply;
- the receptor actor, `recept`, target of the server actor answer messages;
- the client actor that initiates the system: it sends a message to replicate the server actor and then launches a message `m`.

The system is analyzed on a 2GHz Intel Core 2 Duo with 1 Go of RAM under Mac OSX 10.5. It takes 52 seconds to compute the least fixed point in 32 iterations. All the details of computed properties can be found in Chapter B. Let us just highlight some properties that can be observed.

Control flow The control is very precise and does not introduce any over-approximation. No computed transition is spurious.

Boundedness The concrete system is finite as it consider the receiving of a single message `m`. In that particular case, our analyzer is able to infer the best bounds. All the reductions used allow to rely on the occurrences properties both globally and locally to obtain the precise occurrence of message labels in interface and mailbox approximation. Furthermore this precision participate to the accuracy of computed transitions since it forbids some spurious ones.

Linearity The linearity abstraction gives here a `•` value to all address binder. Each address created is then used in safe way, allocating at most one actor on it.

Orphan freeness Finally the orphan freeness can be observed in almost all actors. Except the actor bound on binder 2, all other actors satisfy the orphan freeness property: all their message label can be consumed in their future; furthermore they satisfy the liveness orphan definition since no actor is ever stuck. The only weakness of the result is, as foreseeable, the case of the actors on program point 2. The main mechanism of this system imposes that the server actor dies and send its arguments before being replicated. This step of computation introduces a state where there is no actor bound on the address (a dead node in the interface and mailbox abstraction). The produced message `m` is then associated to the dead node itself or to one of its ancestor. In that case, we cannot guarantee its consumption.

8.3 USAGE

The PACSA prototype is freely available and will be soon released. Analyses can be currently computed using two front-end interfaces. The first one is a command line query and the second one is a web interface to the tool.

Figure 8.7 Command line usage.

```

#:~$ ./pacsa --help
Usage: pacsa [OPTION]... [FILE]...
Compute the set of properties of the term specified in FILE (if FILE
is not given the standard input is used)
  -v verbose level: 0,1 or 2 (default is 0)
  -d debug level: 0,1 or 2 (default is 0)
  -output-format output format: xml or string (default is string)
  -domains abstract domains used to represent properties during the
    analysis: occ, part, mailbox, or mx+occ
  -parse-only parse-only
  -temp-dir Directory where to store temporary files for use with web
    interface
  -base-url Base url for webdot call for use with web interface must
    point to the temp-dir place
  -help Display this list of options
  --help Display this list of options

```

The web interface is available on the author website and the tool source will be soon available here also.

8.3.1 *Command line*

The command line client is the core analyzer. It takes a CAP term either into a file or on the standard input. Among possible arguments, `domains` allows to choose the abstraction used in the computations. It goes from the simple control flow to complex combinations of domains with reductions between them at different levels. The parameter `v` specifies the level of verbose of the analysis result. Choosing higher values gives details of all computed transitions: whether they were effectively computable, whether the fixed point algorithm reached a local fixed point, etc. However, smaller values give only the resulting abstract elements or a set of high level properties that can be inferred from the abstract element, such as boundedness, linearity, or actors and messages sharing the same address.

The principal drawback of such command line interface is the readability of abstract element values. For example, the channel and marker approximation, the shape analysis or the sequence of interfaces abstraction are graphs or automata and are not easily readable.

The Figure 8.7 gives the usage of the tool while Figure 8.8 shows an example of results obtained by the analyzer.

Figure 8.8 Results of CAP term analysis on command line.

```

#:~$ ./pacsa examples/ping_pong.cap -v 1

Analyzed term:
va^1, vb^2,
  a >^3 [ ping^4() = $(e, s) ({ b <^5 pong( ) || e >^6 s }) ]
|| b >^7 [ pong^8() = $(e, s) ({ a <^9 ping( ) || e >^10 s }) ]
|| a <^11 ping( )

Initial state extracted. Computing fixpoint:
.....

Least fixpoint reached after 12 iterations.
The abstract element, least fixpoint of the collecting semantics
of the term, is:

11:
Marker: graph G { 647; } over the variable set: I, a
      mapping from node to partition: , 647 -> I, a,
Channel: graph G { 686; } over the variable set: a
      mapping from node to partition: , 686 -> a,
Automata on the set I, a
I: Initial States: {ens vide}, Final States: {ens vide},
  Transitions : contains the empty word
a: Initial States: {1}, Final States: {1}, Transitions :

10:
Marker: graph G { 645; 646; } over the variable set: I, e, s
      mapping from node to partition: , 645 -> I, 646 -> e, s,
Channel: graph G { 684; 685; } over the variable set: e, s
      mapping from node to partition: , 684 -> e, 685 -> s,
Automata on the set I, e, s
I: Initial States: {10}, Final States: {10},
  Transitions : , 10 -> {10} contains the empty word
e: Initial States: {2}, Final States: {2}, Transitions :
s: Initial States: {7}, Final States: {7}, Transitions :

9:
Marker: graph G { 643; 644; } over the variable set: I, a
      mapping from node to partition: , 643 -> I, 644 -> a,
Channel: graph G { 683; } over the variable set: a
      mapping from node to partition: , 683 -> a,
Automata on the set I, a
I: Initial States: {10}, Final States: {10},
  Transitions : , 10 -> {10} contains the empty word
a: Initial States: {1}, Final States: {1}, Transitions :

```

Figure 8.9 Home page of the web interface of PACSA.

Home CAP calculus Examples Analyzer

PACSA
 PACSA, Primitive Actor Calculus (CAP in french), a Static Analyzer, is a tool based on the abstract interpretation in order to soundly over-approximate the collecting semantics of CAP terms.
 The CAP calculus is a formal language to express high level systems in a formal calculus. It has a well defined syntax and semantics presented in the **CAP calculus** section.
 Abstract interpretation is a formal technique in order to express static analyses in a unified way. It was proposed in the 70s by **Patrick Cousot**. The current analyzer relies on previous works by Arnaud Venet and **Jérôme Feret** on the static analysis of concurrent systems by abstract interpretation, especially applied on Pi calculus or Ambients.

Global Mechanism
 The analyzer computes the following steps:

- It parses the CAP term and expresses in a modified form, easing the analysis.
- It computes a fixed point computation of the collecting semantics of the term, according to the user choice of abstract representation.

The global mechanism is presented in the following figure.

Download
 PACSA is not currently available but will be soon released. Please contact me by email if you want more information or if you want to try the current prototype.

8.3.2 Web interface

The web interface is a much more friendly approach to the static analysis of CAP. The website aims at presenting the problematics of analyzing CAP terms. It presents both CAP syntax and semantics and provides examples. Example terms are given and explained. Finally the last part of the website provides the interface to the tool. One can then copy and paste an example term into the analyzer form and choose the domain that is used to compute the fixed point approximation of the term collecting semantics.

The following Figures illustrate the different pages of this site. The Figure 8.9 gives the home page, describing problematics and presenting the framework overall mechanism. The Figure 8.10 shows the first visible example on the examples section. Figure 8.11 and Figure 8.12 give the result of the analysis of a CAP term. The first one shows the beginning of the analysis log, detailing the different computed abstract transitions. The second one gives the result of the abstract interfaces and mailboxes, computed using the partitioned analysis.

Figure 8.10 Examples descriptions given in the web site.

Home	CAP calculus	Examples	Analyzer
------	--------------	----------	----------

Examples
 We present here some examples of CAP terms. Copying a term and pasting it into the analyzer page allows you to launch the analysis on the selected term.

Linear cell
 The linear cell is an actor that first receive a value v by a message $put(v)$ and then is able to send it back to the address x when receiving a message $get(x)$.

```

~a ~b ~c
a > [
  put ( v ) = $(e.s, v ) {
    e > [get ( y ) = $(e'.s_p)(y < rep(v) || e' > s_v ) ]
  }
]
|| a < put (b)
|| a < get (c)
|| c > [rep(val) = $(e.s)(e > s)]
  
```

Several variants are available with infinite behaviour, finite or infinite mailbox size, etc.
 Infinite behaviour: the get message handling send back two message and initiate an infinite behaviour

```

~x ~a ~b
a > [put(x) = $(e.s) (e > [get(y)=$(f.f)(y < m(x) || e > s || e < put(x) || e < get(y) ])]
|| a < put(x)
|| a < get(b)
  
```

This is a modified version with infinite bahviour but a bounded sized term.

```

~x ~a ~b
a > [put(x) = $(e.s) (e > [get(y)=$(f.f)(y < m(x,e,s))])]
|| a < put(x)
|| a < get(b)
|| b > [m(val,ego,beh) = $(e.s)( ego > beh || ego < put(val) || ego < get(e) || e > s )]
  
```

Ping-pong

Figure 8.11 A first computation showing a log of the analysis.

The Analyzer

```

~x -a -b
a > [put(x) = $(e,s) (
  e > [get(y)=$(f,t)(y < m(x,e,s))]
  a < put(x)
  a < get(b)
  b > [m(val,ego,beh) = $(e,s)( ego > beh
      | ego < put(val)
      | ego < get(e)
      | e > s )]]

```

Abstract Domains
 Occurrence Counting

Results

- Result
- Log
- Control Flow
- Occurrence Counting

Log

The term was correctly parsed
 Term analyzed :
 $Vx^1, Va^2, Vb^3,$
 $a >^4 [put^5(x) = \zeta(e, s) ((e >^6 [get^7(y) = \zeta(f, t) ((y <^8 m(x, e, s))]))]]$
 $ll a <^9 put(x)$
 $ll a <^{10} get(b)$
 $ll b >^{11} [m^{12}(val, ego, beh) = \zeta(e, s) ((ego >^{13} beh \parallel ego <^{14} put(val) \parallel ego <^{15} get(e) \parallel e >^{16} s))]]$

Initial state extracted. Computing fixpoint:

```

(4, 9): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(4, 9): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(4, 14): .constraints?. invalid
(6, 10): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(6, 10): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(6, 15): .constraints?. invalid
(11, 8): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(11, 8): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(5, 13, 9): .constraints?. invalid
(5, 13, 14): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(5, 13, 14): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(5, 16, 9): .constraints?. invalid
(5, 16, 14): .constraints?. invalid
(7, 13, 10): .constraints?. invalid
(7, 13, 15): .constraints?. invalid
(7, 16, 10): .constraints?. invalid
(7, 16, 15): .constraints?. invalid
(12, 13, 8): .constraints?. invalid
(12, 16, 8): .constraints?. invalid
(4, 9): .the local fixpoint is already reached
(4, 14): .constraints?. invalid
(6, 10): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(6, 10): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(6, 15): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. nop
(6, 15): .constraints?. ok, launching continuations?. ok, computing widen?. ok, local fixpoint reached?. yeap
(11, 8): .constraints?. ok, launching continuations?. ok, computing union?. ok, local fixpoint reached?. yeap
(5, 13, 9): .constraints?. invalid
(5, 13, 14): .the local fixpoint is already reached
(5, 16, 9): .the local fixpoint is already reached
(5, 16, 14): .the local fixpoint is already reached
(7, 13, 10): .constraints?. invalid
(7, 13, 15): .constraints?. invalid
(7, 16, 10): .constraints?. invalid
(7, 16, 15): .constraints?. invalid

```

Figure 8.12 A second computation showing resulting abstract elements for the interfaces and mailboxes approximation.

Home	CAP calculus	Examples	Analyzer
------	--------------	----------	----------

The Analyzer

```

-x -a -b
a > [put(x) = $(e,s) (
  e > [get(y)=$(f,t)(y < m(x,e,s))])]
||
a < put(x)
a < get(b)
b > [m(val,ego,beh) = $(e,s)(
  ego > beh
  || ego < put(val)
  || ego < get(e) || e > s )]
```

Abstract Domains
Armageddon Compute

Results

- Result
- Log
- Control Flow
- Occurrence Counting

Term
 $\forall x^1, v a^2, v b^3, a >^4 [put^5(x) = \zeta(e, s) (e >^6 [get^7(y) = \zeta(f, t) (y <^8 m(x, e, s))])] \parallel a <^9 put(x) \parallel a <^{10} get(b) \parallel b >^{11} [m^{12}(val, ego, beh) = \zeta(e, s) (ego >^{13} beh \parallel ego <^{14} put(val) \parallel ego <^{15} get(e) \parallel e >^{16} s)]$

Properties
 Partition for binder [2]
 Mailbox

```

graph TD
    N4((4,A)) -- "(put, 4)" --> N6A((6,A))
    N6A -- "(get, 6)" --> N6D((6,D))
    N6D -- "(put, 5)" --> N6A
    N6D -- "(--, 12)" --> N13A((13,A))
    N13A -- "(--, 11)" --> N6D
    style N4 stroke-width:4px
    
```

CONCLUSION

In this thesis, we proposed to apply abstract interpretation-based techniques to verify properties on an actor based process calculus, CAP. This final chapter summarizes the contributions and describes some the generalization of this work to other calculi or properties.

CAP was first defined in 1996 in [24] and some properties of interest of the calculus were defined, in particular the linearity property and the orphan-freeness property. Since then, multiple works [23, 30, 39] have addressed the development of static analyses for these properties using $HM(X)$ based type inference approaches. Each of these analyses faced difficulties when checking the aforementioned properties. Let us enumerate the principal problems encountered.

First, the CAP behavior passing capability makes it difficult to analyze. Theoretical results [100] state that it is not more expressive than first order calculi, but defining meaningful types for a higher-order calculus is intricate and we are not aware of any proposal yet.

Second, it is not easy to represent numerical values within types. Properties that rely on a numerical representation, such as the number of available messages in the orphan message detection analysis, are not easily representable. Computing a sound type requires a widening step. And the use of the higher-order feature of CAP renders the analysis inapplicable: the only type system that dealt with this behavior passing mechanism uses a finite domain for abstract variable values (the usage mode).

Third, typing rules usually do not handle dead code well. The full term is typed and sub-parts that will never be used may break the desired property. Solutions to this particular problem involve costly fixed point typing of a term or the use of dependent types which resolution is exponential in time, when an algorithm exists. Furthermore, even if the decidability of the constraints checking is provided, the existence of a principal type is an important concern to address.

Finally type systems are not easily defined. The typing rules and the generated constraints mix the information related to the control flow of terms, expressing which parts can interact together, and the information specific to the considered property. Furthermore a small modification of the property definition requires a full proof of the typing soundness from the beginning. It is not usually extensible in a smooth way.

Another more recent approach for the static analysis of concurrent models was first defined by VENET [107] and later generalized by FERET [49]. These works address the over-approximation of the collecting semantics of process calculi terms, allowing to observe properties of concurrent systems.

In order to precisely approximate terms, analyzed calculi have first to be encoded in a non standard semantics that allows to make explicit the history of transitions leading to creation of threads and values. This encoding avoids the use of α -conversion when manipulating terms and gives more available information to the applied abstractions.

Once the term is expressed in the non standard form, its collecting semantics can be over-approximated within the framework of abstract interpretation. Abstract domains are then used to approximate the control flow information or occurrence counting properties of reachable configurations. These abstract domain elements represent properties verified by a set of non standard configurations related by a concretization function. The associated abstract transition allows to represent on the abstract elements the effect of computing transitions on the set of related configurations.

The methodology of abstract interpretation gives the necessary condition in the domain definitions in order to obtain sound and decidable abstractions of reachable terms. It also allows to combine abstractions, facilitating the improvement of analyses accuracy.

9.1 CONTRIBUTIONS

This current work is at the crossroad between the two preceding concerns. In order to solve the difficulties encountered by the typing-based static analysis of CAP, we have switched to the abstract interpretation approach, targeting the same properties.

The contributions of this thesis can be summarized as follows:

- We express CAP in a non standard form, making explicit the creation of threads and values. We proved the soundness of the encoding by exhibiting a strong bisimulation result between the standard CAP semantics and its non standard encoding.
- We adapt existing abstract domains and introduce new ones to verify properties specific to CAP. In particular, we enhance the occurrence counting abstraction in order to avoid some spurious transitions. We introduce a new partitioned analysis devoted to the representation of properties specific to a given CAP address. This abstraction is essential when analyzing CAP, since the notion of address is central in CAP and in its peculiar properties.

- We introduce an abstract domain that allows to check the linearity property of a term. This property expresses that no address is used twice or more in the same configuration to bind an actor. The proposed abstract domain relies on a backward flow computation to infer the usage of variables in messages depending on the value already inferred in the future threads. It represents information specific to the linearity property and relies on other abstract domains to over-approximate feasible transitions. It has therefore no constraints on the form of analyzed terms and fully handle the CAP behavior passing ability.
- Finally we propose an analysis allowing to guarantee the absence of orphan messages. In that context, orphans are messages that are sent to an address that will never be able to handle them. Preceding analyses lacked a precise control flow computation and did not allow behavior passing. The proposal is to check the property in two phases.

Using the abstraction framework for CAP, presented above, a Vector Addition System with States (VASS) is computed for each address. It represents the sequences of actors associated to a given address value as well as the evolution of its available messages. The orphan-freeness property is then expressed as a coverability problem on obtained VASSs.

The second phase aims at approximating the mailbox, *i.e.* the set of messages available for an actor, at each step of its computation using HIGMAN lemma or a fixed point expression. We then ensure that each message label can be received in a sound over-approximation of all its future maximal paths.

All the different contributions have been experimented in the PACSA tool presented in Chapter 8.

The works presented here are another contribution to the practical deployment of abstract interpretation to concurrent calculi. The application of this approach of static analysis by abstract interpretation to CAP allows to solve all difficulties encountered in previous type inference based analyses [25, 26, 27, 28, 29], in particular concerning the precision of the analysis, the capability to handle behavior passing, and finally the methodology of designing new analyses provided old ones.

The author advocates that, in general, this approach to the static analysis of concurrency has many advantages compared to the other methodologies.

Its soundness-by-construction and its capability to construct new abstractions by computing products or reductions of existing domains allows to easily define analyses of new properties without proving everything from scratch and taking benefit from other domains properties and accuracy.

A second main advantage is the precision obtained when abstracting control flow of the non standard encoding. All other approaches lack a fine representation of it and lead to the analysis of dead branches.

Finally the use of relational abstraction is easily applicable in this general context of abstract interpretation and allows to represent non trivial properties between different values or thread instances. It is essential when one needs to differentiate recursive instances of values or threads and opens the road to the analysis of higher level properties.

9.2 FUTURE WORKS

Many perspectives are opened by the current work. We now present five of them at short or middle term:

- at short term, we target to
 - extend the implementation of PACSA to implement the HIGMAN lemma and allow the full analysis of the orphan message absence;
 - enhance the linearity abstraction to deal more precisely with internal free variables;
- at middle term, we would like to
 - apply the abstraction of causality, based on a VASS representation, to the π -calculus;
 - analyze other paradigms of concurrency, low level ones like process calculi to more high level ones such as the Erlang language;
 - combine abstract interpreters using a weaving mechanism, allowing to define and analyze more realistic concurrent systems, developed with different programming paradigms.

9.2.1 *Implementing mailboxes over-approximation*

The current implementation of the orphan-freeness checking only handles the first phase, providing a VASS representation of actor behavior sequences for each address.

A short term perspective would be to integrate the over-approximation of mailboxes within the PACSA prototype. An implementation relying on the HIGMAN lemma would consist of a depth-first search algorithm, widening growing occurrences of message labels in mailboxes. The full orphan-freeness checking could then be computed by the tool.

9.2.2 *Introducing relational abstraction in the linearity abstract domain*

The current linearity abstract domain associates an abstract representation of the number of possible actors installed on a value at each thread environment variable. However, in case of internal free variable, nothing guarantees us that the free variable associated to the installation of an actor is not used recursively more than once. Introducing relational information or a reduction between the occurrence counting abstraction and the linearity one could straighten the resulting properties, and avoid possible false alarms.

9.2.3 *Applying the proposed domains to the analysis of π -calculus*

For many years, KOBAYASHI has been developing type based analyses devoted to complex properties checking for the π -calculus. Recently he proposed in [74] a type inference-based approach of his past works. This type system intends to assign to each channel a usage denoting the read and write behavior associated to it. This allows him to verify properties, such as deadlock-freeness.

These works are very interesting as they consider raw terms without any annotation and allow to automatically infer high level properties. However it undergoes usual type systems difficulties. For example, it does not deal well with recursive instances of a thread associated to different usages, since it does not use polymorphic or dependent types.

A very challenging perspective would be to compare the results we obtain with our abstraction of interfaces, defined for the orphan-freeness checking, with these works on the π -calculus. The π -calculus is already expressed within this framework of abstract interpretation for process calculi. So the challenge would be to adapt the interface approximation under the partitioned abstract domain to represent meaningful properties for π -calculus terms.

The extension could be defined in two steps. We first need to adapt the linearity abstraction to count not only possible reads (an actor installed to an address being understood as a reading thread for this address) but also possible writes. It would necessitate a more complex underlying numerical abstraction than the one presented in the linearity property checking, such as the interval one for example.

Then a second phase would be to combine our abstraction of causality, associating a VASS to each channel, with the occurrence counting abstraction of reads and writes as given by the linearity domain extension.

9.2.4 *Analyzing other kinds of concurrent communicating models*

We advocate that the current approach of the static analysis of concurrency, based on an abstract interpretation of an encoded version of concurrent cal-

culi or languages, is very promising to analyze other formal calculi but also more high level and realistic languages. We outline here two possible ways of extensions. The first one is another application to process calculi devoted to the modeling of service-oriented systems and the second one is the application to more high level descriptions, such as the Erlang language.

Concurrent calculi for service-oriented systems

Nowadays new process calculi are still defined to model specific systems. An active branch is the community that considers modern service-oriented systems or their future evolutions. In the current EU-project Sensoria [1], multiple process algebras are being developed in order to understand the right notions of services in a global net. This includes process algebras like COWS [78], focusing on various forms of session-based interactions or more general correlation-based interactions.

Session-based analyses [98, 99] are often based on type checking systems where the program or the term is annotated by the end-user with a type describing the targeted specification. The approach proposed in this thesis could also be applied in such a context, providing precise results. However it would be necessary to express unusual high level mechanisms that can be found in these calculi, like specific scope definitions or global mechanisms to kill threads.

More realistic languages like ERLANG

An interesting high level language target for applying this framework of concurrency analysis is the Erlang language. Erlang is an industrial functional language developed by Ericsson for its communication software. It has built-in support for concurrency, distribution and fault tolerance. Its concurrency model is very similar to the actor one. Processes are associated to an address and to a mailbox containing their pending messages. Static analyses for CAP have already been applied to Erlang in [39, 40] but as for CAP, relying on our abstract interpretation framework could give even more precise results.

Currently, existing available tools only consider low level properties. Applying the proposed approach to the Erlang language could provide useful tools to this telecommunication community.

9.2.5 *Weaving abstract interpretations*

This last perspective considers the combination of abstract interpretations. It comes from the idea of the Aspect-oriented programming (AOP) community where programs or systems are specified in separate descriptions weaved together using a weaving mechanism [69].

The main idea is the following: a program is composed of two or more descriptions with a formal model to weave them together. The AOP *à la* AspectJ

is one of such formal models based on Java descriptions where one of the description is special and denotes the base program which is executable, then others descriptions denote aspects and are tangled using the AspectJ weaver. The elements of the base program and of the aspects are woven by matching expressions on their program points or internal values in a more or less dynamic manner using a weaving algorithm.

Working on a calculus describing an abstraction of a system while only representing the communication between processes, we consider that a concurrent system could be fully defined using two descriptions: a first one describing its concurrent or even distributed behavior, and a second one specifying its functional behavior.

Such a description could then be analyzed by taking profit of existing analyses devoted to each description paradigm. This framework of combining abstractions could then be extended in order to be applied to existing Aspect frameworks like AspectJ.

Once the global semantics of the woven system has been defined using underlying aspects semantics, it can be over-approximated using specific abstract interpretations. Approximating the collecting semantics of the woven program happens at different steps of the abstract computation.

First, when computing an abstract transition of a semantics in the woven program analysis, we check whether the information in the current abstraction of the concrete state representation allows such a transition. The approximation of such information in the abstract part of the abstract state related to this semantics leads to an over-approximation of all really applicable transitions.

A second level of approximation occurs during the abstract weaving. The first step of the abstract transition computation is satisfied and the transition can be computed. But the matched transition may call another semantics during its computation. We therefore have to compute an over-approximation of all matchable joinpoints in the current abstract transition.

Finally, both kinds of approximations accumulate when the control-flow is modified and passes back and forth from one semantics to another.

These different issues have to be addressed to provide a framework for approximating aspect oriented programs. Depending on the underlying paradigms and on the description of the weaving, more or less complex abstractions would have to be combined.

This perspective is, in our opinion, very promising, since it allows to rely on existing abstractions to define very high level properties on complex systems. Furthermore, it provides a powerful and extensible approach to the analysis of aspects on which there is, nowadays and in our knowledge, no practical answer.

INDEX OF NOTATIONS

(p, id, E)	A thread denoting an active part of an encoded CAP term; defined by the program point p and the identity marker id , and associated with the environment E .	28
(p^a, id^a, E^a)	Interacting actor thread in a tuple of interacting threads $(p^k, id^k, E^k)_k$.	59
(p^m, id^m, E^m)	Interacting message thread in a tuple of interacting threads $(p^k, id^k, E^k)_k$.	59
Π	Translation function mapping CAP non standard configurations to standard ones.	40, 200
Σ	The set of transition labels built over \mathcal{L}_p .	41
\triangleright	Actor definition.	8
β	Extraction function building sets of static threads from CAP subterms.	33, 35
\mathcal{V}_c^∞	The subset of \mathcal{V}_c denoting variables associated to an unbounded number of thread or transition occurrences.	91
\triangleleft	Message definition.	8
ν	Binder definition.	8
$\psi(u)$	The abstraction of a transition label to one of its program point label.	49
\odot	This special program point is used in the abstraction of interface sequences used to ensure orphan messages absence. It is associated to edges from F nodes to T ones and denotes no message consumption. It is also used in the ancestor computation where it denotes no dependencies.	143, 145, 146
\mathcal{A}	The set of partial interaction names.	31
$behavior_n$	Partial interaction type denoting a behavior branch binding n variables.	31
\mathcal{C}	The set of non standard configurations, <i>i.e.</i> sets of (p, id, E) triples.	41
C_{beh}	The subset of a CAP non standard configuration containing only behavior definition threads.	58

$C _x$	The subset of a CAP non standard configuration considering only actor and message associated with the address x .	58
dynamic_actor	Partial interaction type denoting an actor whose behavior is defined by a variable.	31
I	The interface function I maps program points in \mathcal{L}_p to their associated set of variables, the domain of their threads environment.	44
I^k	Variable denoting the identity marker of the k -th interacting thread in CAP non standard formal rules.	32
\mathcal{H}	The abstract domain of affine equality, the Karr domain.	52
\mathcal{L}	The set of program points.	8
\mathcal{L}_a	The set of actor program points.	8
\mathcal{L}_b	The set of behavior branch program points.	8
\mathcal{L}_m	The set of message program points.	8
\mathcal{L}_v	The set of binder program points.	8
\mathcal{L}_p	The set of CAP active parts (actors, messages and behavior branches).	8
\mathcal{M}	The set of identity markers.	28, 29
message _n	Partial interaction type denoting a message with n arguments.	31
$m_i(\tilde{x}_i) = \zeta(e_i, s_i)C_i$	Behavior branch definition.	8
\mathcal{M}_l	The set of message labels.	8
\mathcal{N}	The set of actor names, ie address.	8
$\mathcal{N}_{\mathcal{V}_c}$	The abstraction mapping sets of occurrences as built by $\mathbb{N}^{\mathcal{V}_c}$ to numerical abstraction of vector of nonnegative integers indexed by \mathcal{V}_c .	47
$\mathbb{N}^{\mathcal{V}_c}$	The abstraction mapping a configuration to the occurrence of each thread and transition label as indexed by \mathcal{V}_c .	47
pi	Partial interaction definition pi = (s, (parameter), (bound), continuation) with a type, a set of parameter variables, a set of bound variables and a continuation, a set of static threads.	33

\mathcal{R}	Formal rule definition used in (abstract) transition definitions $\mathcal{R} = (n, \text{components}, \text{compatibility}, v_passing)$ with n the number of required partial interactions, components their associated type, compatibility the equality requirements among interacting threads variables, and $v_passing$ the value passing induced by the transition.	32, 37
static_actor_n	Partial interaction type denoting a behavior branch binding n variables of a syntactically defined actor.	31
\mathcal{V}	The set of variables.	8
VASS	Vector Addition System with States.	137
\mathcal{V}_c	The set of variables denoting occurrences of threads and transition labels, $\mathcal{V}_c = \mathcal{L} \times \mathbb{B}$.	47
X_i^k	Variable denoting the i -th parameter of the k -th interacting thread in CAP non standard formal rules.	32
Y_j^k	Variable denoting the j -th bounded variable of the k -th interacting thread in CAP non standard formal rules.	32

BIBLIOGRAPHY

- [1] Sensoria project. URL <http://www.sensoria-ist.eu/>. (Cited on page 196.)
- [2] Gul Agha. Hal: A high-level actor language and its distributed implementation. In *the 21st International Conference on Parallel Processing (ICPP'92)*, volume 2, pages 158–165, August 1992. (Cited on page 21.)
- [3] Gul Agha and Carl Hewitt. Actors: a conceptual foundation for concurrent object-oriented programming. *Research directions in object-oriented programming*, pages 49–74, 1987. (Cited on page 20.)
- [4] Gul Agha, I. Mason, Scott F. Smith, and Carolyn Talcott. Towards a theory of actor computation. In *CONCUR*, volume 630 of *LNCS*. Springer, 1992. (Cited on page 20.)
- [5] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993. (Cited on page 35.)
- [6] Alexander Aiken, Edward L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *POPL*, pages 163–173, New York, NY, 1994. (Cited on page 35.)
- [7] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded c programs. In K. Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 35–51. Springer, 2006. (Cited on page 34.)
- [8] Roberto M. Amadio and Ch. Meyssonier. On the decidability of fragments of the asynchronous pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 52(1), 2001. (Cited on page 37.)
- [9] Roberto M. Amadio and Charles Meyssonier. On decidability of the control reachability problem in the asynchronous pi-calculus. *Nordic Journal of Computing*, 9(2):70–101, 2002. (Cited on page 173.)
- [10] Roberto M. Amadio, Gérard Boudol, and Cédric Lhousseine. On message deliverability and non-uniform receptivity. *Fundam. Inform.*, 53(2):105–129, 2002. (Cited on page 36.)
- [11] Jean-Marc Andreoli, Tiziana Castagnetti, and Remo Pareschi. Abstract interpretation of linear logic programming. In *ILPS*, pages 295–314, 1993. (Cited on page 37.)

- [12] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In R. Alur and D. Peled, editors, *CAV*, volume 3114 of *LNCS*, pages 484–487. Springer, 2004. (Cited on page 36.)
- [13] C. Baier and H. Hermanns, editors. *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *LNCS*, 2006. Springer. (Cited on pages 204, 208, and 209.)
- [14] Thomas Ball, Sagar Chaki, and Sriram K. Rajamani. Parameterized verification of multithreaded software libraries. In Margaria and Yi [81], pages 158–173. (Cited on page 172.)
- [15] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In D. Sangiorgi and R. de Simone, editors, *CONCUR*, volume 1466 of *LNCS*, pages 84–98. Springer, 1998. (Cited on page 36.)
- [16] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis for the pi-calculus with applications to security. *Inf. Comput.*, 168(1):68–92, 2001. (Cited on page 36.)
- [17] Chiara Bodei, Pierpaolo Degano, Hanne Riis Nielson, and Flemming Nielson. Flow logic for Dolev-Yao secrecy in cryptographic processes. *Future Generation Computer Systems*, 18(6):747–756, 2002. (Cited on page 36.)
- [18] Gérard Boudol. Typing the use of resources in a concurrent calculus (extended abstract). In R. K. Shyamasundar and K. Ueda, editors, *ASIAN*, volume 1345 of *LNCS*, pages 239–253. Springer, 1997. (Cited on page 36.)
- [19] Gérard Boudol. The pi-calculus in direct style. In *POPL*. ACM, 1997. (Cited on page 21.)
- [20] Roberto Bruni, Hernán C. Melgratti, and Ugo Montanari. Event structure semantics for nominal calculi. In Baier and Hermanns [13], pages 295–309. (Cited on page 37.)
- [21] Rubén Carvajal-Schiaffino, Giorgio Delzanno, and Giovanni Chiola. Combining structural and enumerative techniques for the validation of bounded petri nets. In Margaria and Yi [81], pages 435–449. (Cited on page 173.)
- [22] C. K. Chiu and J. H. M. Lee. Interval linear constraint solving using the preconditioned interval gauss-seidel method. In *ICLP*, pages 17–31, 1995. (Cited on page 105.)

- [23] Jean-Louis Colaço. *Analyses Statiques de Langages d'Acteurs par inférence de types*. Thèse de doctorat, ENSEIHT, Toulouse, 1997. (Cited on pages 5, 14, and 191.)
- [24] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. An actor dedicated process calculus. In *the ECOOP'96 Workshop on Proof Theory of Concurrent Object-Oriented Programming (PTCOOP'96)*, 1996. (Cited on pages 5, 14, 20, 21, and 191.)
- [25] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. A set-constraint-based analysis of actors. In *FMOODS*, pages 107–122. Chapman & Hall, 1997. (Cited on pages 36, 171, and 193.)
- [26] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Analyse de linéarité par typage dans un calcul d'acteurs. In *Actes des Journées Francophones des Langages Applicatifs*, 1997. (Cited on pages 36, 127, 140, and 193.)
- [27] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. From set based to multiset based analysis: A practical approach. In *Workshop on Set Constraints and Constraints Based Program Analysis (CP'98)*, pages 1–10. Pise Univ., 1998. (Cited on pages 36, 171, and 193.)
- [28] Jean-Louis Colaço, Marc Pantel, Fabien Dagnat, and Patrick Sallé. Static safety analysis for non-uniform service availability in actors. In *FMOODS*, volume 139, pages 371–386. Kluwer, 1999. (Cited on pages 36, 171, and 193.)
- [29] Jean-Louis Colaço, Marc Pantel, and Patrick Sallé. Static analysis of behavior changes in actor languages. In J.-P. Bahsoun, T. Baba, J.-P. Briot, and A. Yonezawa, editors, *Object-Oriented Parallel and Distributed Programming*, pages 53–72. Hermès, 8, quai du Marché-Neuf, 75004 Paris, France, 2000. (Cited on pages 36, 171, and 193.)
- [30] Matthias Colin. *Analyse statique de la communication dans un langage d'acteurs fonctionnel*. Thèse de doctorat, Institut National Polytechnique, Toulouse, France, 2002. (Cited on pages 5, 14, and 191.)
- [31] Matthias Colin, Xavier Thirioux, and Marc Pantel. Temporal logic based static analysis for non uniform behaviors. In *FMOODS*, number 2884 in LNCS, pages 94–108. Springer, 2003. (Cited on pages 171 and 172.)
- [32] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999. (Cited on page 32.)
- [33] Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod. (Cited on page 34.)

- [34] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977. (Cited on pages 3, 8, 19, 31, and 32.)
- [35] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282. ACM, 1979. (Cited on pages 8 and 32.)
- [36] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992. (Cited on page 33.)
- [37] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97. ACM, 1978. (Cited on page 34.)
- [38] Régis Cridlig. Semantic analysis of concurrent ml by abstract model-checking. *Electr. Notes Theor. Comput. Sci.*, 5, 1996. (Cited on page 36.)
- [39] Fabien Dagnat. *Vérification statique de programmes répartis*. Thèse de doctorat, Institut National Polytechnique de Toulouse, Toulouse, 2001. (Cited on pages 5, 14, 191, and 196.)
- [40] Fabien Dagnat and Marc Pantel. Static analysis of communications for erlang. In *the 8th International Erlang/OTP User Conference*. Ericsson Telecommunication, 2002. (Cited on page 196.)
- [41] Fabien Dagnat, Marc Pantel, Matthias Colin, and Patrick Sallé. Typing concurrent objects and actors. In *L’Objet - Méthodes formelles pour les objets*, volume 6,1, pages 83–106. Hermès, 2000. (Cited on page 171.)
- [42] Silvano Dal Zilio. *Le calcul bleu: types et objets*. PhD thesis, Université de Nice - Sophia-Antipolis, 1999. (Cited on page 36.)
- [43] M. Dam, editor. *Analysis and Verification of Multiple-Agent Languages, 5th LOMAPS Workshop, Stockholm, Sweden, June 24-26, 1996, Selected Papers*, volume 1192 of *LNCS*, 1997. Springer. (Cited on page 211.)
- [44] Giorgio Delzanno. On model checking for a nominal process calculus, 2003. (Cited on page 37.)
- [45] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond -limiting. In *PLDI*, pages 230–241, 1994. (Cited on page 34.)
- [46] Joubine Dustzadeh and Elie Najm. Consistent semantics for odp information and computational models. In A. Togashi, T. Mizuno, N. Shiratori, and T. Higashino, editors, *FORTE*, volume 107 of *IFIP Conference Proceedings*, pages 107–126. Chapman & Hall, 1997. (Cited on page 35.)

- [47] Jérôme Feret. Occurrence counting analysis for the pi-calculus. In *Geometry and Topology in CONcurrency Theory (GETCO'01)*, volume 39.2 of *ENTCS*. Elsevier, 2001. (Cited on page 63.)
- [48] Jérôme Feret. Mobile system thread partitioning. (personnal communication), 2007. (Cited on page 96.)
- [49] Jérôme Feret. *Analysis of Mobile Systems by Abstract Interpretation*. PhD thesis, École polytechnique, Paris, France, 2005. (Cited on pages 4, 6, 7, 8, 11, 14, 37, 38, 41, 43, 50, 53, 54, 56, 57, 63, 65, 66, 72, 96, 99, 100, 105, and 192.)
- [50] Jérôme Feret. Confidentiality analysis of mobile systems. In *SAS*, number 1824 in *LNCS*. Springer, 2000. (Cited on page 37.)
- [51] Jérôme Feret. Abstract interpretation-based static analysis of mobile ambients. In *SAS*, number 2126 in *LNCS*. Springer, 2001. (Cited on page 37.)
- [52] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy. A calculus of mobile agents. In *CONCUR*, *LNCS*. Springer, 1996. (Cited on page 21.)
- [53] Pierre-Loïc Garoche. Static analysis of actors by abstract interpretation. Master's thesis, École Normale Supérieure de Cachan, 2005. (Cited on pages 10 and 41.)
- [54] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static analysis of actors: From type systems to abstract interpretation. In F. Ranzato, editor, *1st International Workshop on Emerging Applications of Abstract Interpretation (EAAI'06)*, *ETAPS'06 satellite event, Vienna, Austria*, 26 mars 2006. (Cited on page 123.)
- [55] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Static safety for an actor dedicated process calculus by abstract interpretation. In R. Gorrieri and H. Wehrheim, editors, *FMOODS*, volume 4037 of *LNCS*, pages 78–92. Springer, 14-16 june 2006. (Cited on page 41.)
- [56] Pierre-Loïc Garoche, Marc Pantel, and Xavier Thirioux. Abstract interpretation-based static safety for actors. *Journal of Software (JSW)*, 2 (3):87–98, September 2007. ISSN 1796-217X. (Cited on page 41.)
- [57] Simon J. Gay. A sort inference algorithm for the polyadic pi-calculus. In *POPL*, pages 429–438, 1993. (Cited on page 35.)
- [58] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *J. ACM*, 39(3):675–735, 1992. (Cited on page 173.)

- [59] Eric Goubault. Static analyses of the precision of floating-point operations. In P. Cousot, editor, *SAS*, volume 2126 of *LNCS*, pages 234–259. Springer, 2001. (Cited on page [34](#).)
- [60] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *the 3rd International Joint Conference on Artificial Intelligence (IJCAI'73)*, 1973. (Cited on page [20](#).)
- [61] Graham Higman. Ordering by divisibility in abstract algebras. *London Mathematical Society*, 3(2(7)):326–336, 1952. (Cited on page [165](#).)
- [62] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. (Cited on page [20](#).)
- [63] Frank Huch. Verification of erlang programs using abstract interpretation and model checking. *SIGPLAN Not.*, 34(9):261–272, 1999. (Cited on page [36](#).)
- [64] Atsushi Igarashi and Naoki Kobayashi. A generic type system for the pi-calculus. *Theor. Comput. Sci.*, 311(1-3):121–163, 2004. (Cited on pages [140](#) and [172](#).)
- [65] Suresh Jagannathan and Stephen Weeks. Analyzing stores and references in a parallel symbolic language. In *LISP and Functional Programming*, pages 294–305, 1994. (Cited on page [37](#).)
- [66] H. B. M. Jonkers. Abstract storage structures. In de Bakker and van Vliet, editors, *Algorithmic Languages*, pages 321–343, 1981. (Cited on page [37](#).)
- [67] Richard M. Karp and Raymond E. Miller. Parallel program schemata. *J. Comput. Syst. Sci.*, 3(2):147–195, 1969. (Cited on pages [150](#) and [164](#).)
- [68] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133 – 151, 1976. (Cited on pages [34](#), [63](#), [66](#), [100](#), [102](#), and [182](#).)
- [69] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECOOP*, pages 220–242, 1997. (Cited on page [196](#).)
- [70] Naoki Kobayashi. A new type system for deadlock-free processes. In Baier and Hermanns [[13](#)], pages 233–247. (Cited on page [36](#).)
- [71] Naoki Kobayashi. A partially deadlock-free typed process calculus. In *LICS*, pages 128–139, 1997. (Cited on page [36](#).)
- [72] Naoki Kobayashi. Type-based information flow analysis for the pi-calculus. *Acta Inf.*, 42(4-5):291–347, 2005. (Cited on page [172](#).)

- [73] Naoki Kobayashi. A type system for lock-free processes. *Inf. Comput.*, 177(2):122–159, 2002. (Cited on page 172.)
- [74] Naoki Kobayashi. A new type system for deadlock-free processes. In Baier and Hermanns [13], pages 233–247. (Cited on pages 172 and 195.)
- [75] Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programming. In *OOPSLA*, pages 31–45, 1994. (Cited on pages 21 and 36.)
- [76] Naoki Kobayashi, Motoki Nakade, and Akinori Yonezawa. Static analysis of communication for asynchronous concurrent programming languages. In A. Mycroft, editor, *SAS*, volume 983 of *LNCS*, pages 225–242. Springer, 1995. (Cited on pages 36 and 140.)
- [77] Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the Pi-Calculus. *ACM Trans. Program. Lang. Syst.*, 21(5):914–947, 1999. (Cited on page 140.)
- [78] Alessandro Lapadula, Rosario Pugliese, and Francesco Tiezzi. A calculus for orchestration of web services. In R. de Nicola, editor, *ESOP*, volume 4421 of *LNCS*, pages 33–47. Springer, 2007. (Cited on page 196.)
- [79] Guy Lapaille and Patrick Sallé. Plasma-ii: an actor approach to concurrent programming. *SIGPLAN Not.*, 24(4):81–83, 1989. (Cited on page 21.)
- [80] Francesco Logozzo. *Modular Static Analysis of Object Oriented Languages*. PhD thesis, École polytechnique, Paris, France, june 2004. (Cited on page 34.)
- [81] T. Margaria and W. Yi, editors. *Tools and Algorithms for the Construction and Analysis of Systems, 7th International Conference, TACAS 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*, volume 2031 of *LNCS*, 2001. Springer. (Cited on page 204.)
- [82] Robin Milner. *Communication and Concurrency*. Prentice-Hall, 1989. (Cited on page 20.)
- [83] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006. (Cited on page 34.)
- [84] Ugo Montanari and Marco Pistore. Concurrent semantics for the pi-calculus. *Electr. Notes Theor. Comput. Sci.*, 1, 1995. (Cited on page 36.)
- [85] Elie Najm and Abdelkrim Nimour. Explicit behavioral typing for object interfaces. In A. M. D. Moreira and S. Demeyer, editors, *ECOOP Workshops*, volume 1743 of *LNCS*, page 321. Springer, 1999. (Cited on page 35.)

- [86] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Infinite types for distributed object interfaces. In *FMOODS*, volume 139. Kluwer, 1999. (Cited on page 35.)
- [87] Elie Najm, Abdelkrim Nimour, and Jean-Bernard Stefani. Guaranteeing liveness in an object calculus through behavioural typing. In J. Wu, S. T. Chanson, and Q. Gao, editors, *FORTE*, volume 156 of *IFIP Conference Proceedings*, pages 203–221. Kluwer, 1999. (Cited on page 35.)
- [88] Flemming Nielson, Hanne Riis Nielson, and Helmut Seidl. Cryptographic analysis in cubic time. *Electr. Notes Theor. Comput. Sci.*, 62(1):7–23, 2002. (Cited on page 36.)
- [89] Hanne Riis Nielson and Flemming Nielson. Static and dynamic processor allocation for higher-order concurrent languages. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT*, volume 915 of *LNCS*, pages 590–604. Springer, 1995. (Cited on page 140.)
- [90] Hanne Riis Nielson and Flemming Nielson. Shape analysis for mobile ambients. *Nordic Journal of Computing*, 8:233–275, 2001. (Cited on page 36.)
- [91] Hanne Riis Nielson and Flemming Nielson. Higher-order concurrent programs with finite communication topology. In *POPL*, pages 84–97, 1994. (Cited on page 140.)
- [92] Hanne Riis Nielson and Flemming Nielson. Flow logic: A multi-paradigmatic approach to static analysis. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation*, volume 2566 of *LNCS*, pages 223–244. Springer, 2002. (Cited on pages 35 and 36.)
- [93] Oscar Nierstrasz. Regular types for active objects. In *OOPSLA*, pages 1–15, 1993. (Cited on page 35.)
- [94] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1999. (Cited on page 35.)
- [95] Catuscia Palamidessi, Vijay A. Saraswat, Frank D. Valencia, and Björn Victor. On the expressiveness of linearity vs persistence in the asynchronous pi-calculus. In *LICS*, pages 59–68. IEEE Computer Society, 2006. (Cited on page 140.)
- [96] Benjamin C. Pierce and Davide Sangiorgi. Typing and subtyping for mobile processes. *Mathematical Structures in Computer Science*, 6(5):409–453, 1996. (Cited on page 35.)

- [97] Franz Puntigam. *Concurrent Object-Oriented Programming with Process Types*. Habilitationsschrift, Osnabrück, Germany, 2000. (Cited on page 36.)
- [98] Franz Puntigam. Types for active objects based on trace semantics. In E. Najm, editor, *FMOODS*, Paris, France, 1996. Chapman & Hall. (Cited on page 196.)
- [99] Antonio Ravara and Vasco Thudichum Vasconcelos. Behavioral types in a calculus of concurrent objects. In *the 4th European Conference on Parallel Processing (Euro-Par'97)*, volume 1300 of *LNCS*. Springer, 1997. (Cited on pages 36 and 196.)
- [100] Davide Sangiorgi. From pi-calculus to higher-order pi-calculus - and back. In M.-C. Gaudel and J.-P. Jouannaud, editors, *TAPSOFT*, volume 668 of *LNCS*, pages 151–166. Springer, 1993. (Cited on page 191.)
- [101] David A. Schmidt. Abstract interpretation of small-step semantics. In Dam [43], pages 76–99. (Cited on page 37.)
- [102] Élodie-Jane Sims. Extending separation logic with fixpoints and postponed substitution. *Theor. Comput. Sci.*, 351(2):258–275, 2006. (Cited on page 34.)
- [103] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955. (Cited on page 31.)
- [104] Vasco Thudichum Vasconcelos and M. Tokoro. A typing system for a calculus of objects. In *the 1st International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, volume 742 of *LNCS*. Springer, 1993. (Cited on page 36.)
- [105] Arnaud Venet. Abstract interpretation of the pi-calculus. In Dam [43], pages 51–75. (Cited on page 37.)
- [106] Arnaud Venet. Abstract cofibered domains: Application to the alias analysis of untyped programs. In R. Cousot and D. A. Schmidt, editors, *SAS*, volume 1145 of *LNCS*, pages 366–382. Springer, 1996. (Cited on page 37.)
- [107] Arnaud Venet. *Static Analysis of Dynamic Graph Structures in Untyped Languages*. PhD thesis, École polytechnique, Paris, France, december 1998. (Cited on pages 14, 37, and 192.)

Part III

APPENDICES

PROOFS

A.1 BISIMULATION BETWEEN CAP SEMANTICS AND ITS NON STANDARD ENCODING

We need to prove the correspondence between CAP semantics and its expression in the meta language.

We first formally define the mapping from CAP non configurations to CAP standard configurations. A first step introduces a lemma of well-formedness of CAP non standard terms. Then the main theorem is proved. It is a usual strong bisimulation proof.

Translation

In the following, pi denotes a partial interaction.

Lemma A.1 (non standard term well-formedness) *Let C be a non standard term. C is a set of triples $\{(p, id, E)\}$ where $p \in \mathcal{L}_p$ denotes a program point, $id \in \mathcal{M}$ a marker and $E \in \wp(\mathcal{V} \rightarrow (\mathcal{L} \times \mathcal{M}))$ its environment. Let interaction be the partial map from program points to partial interactions. Let $behavior_set$ be the partial map defined upon the term defined as in 3.2.3 which maps a value denoting a syntactically defined actor program point to its set of behavior program points.*

Every term C is well formed, that is

1. $\forall (p, id, E) \in C$, $interaction(p) \neq \emptyset$ and $\forall (name, var, param, cont) \in interaction(p)$, $\forall v \in var$, $E(v)$ is defined.
2. $\forall (p, id, E) \in C$, such that $\forall pi \in interaction(p)$, pi is a $static_actor$ partial interaction, then $|var_{pi}| = 2$ and there is, in the system, exactly one thread (p_i, id, E_i) for each associated behavior program point p_i .
Each of those threads must exhibit a $behavior_n pi$.
3. $\forall (p, id, E) \in C$, such that $\exists! pi \in interaction(p)$ and pi is a $dynamic_actor$ partial interaction, then $|var_{pi}| = 2$ and let s be the 2nd variable of var_{pi} . There is, in the system, exactly one thread $(p_i, snd(E(s)), E_i)$ for each $p_i \in behavior_set(E(s))$.
Each of those partial interactions must exhibit a $behavior_n pi$.
4. $\forall (p, id, E)$, such that $interaction(p)$ is a $behavior_n pi$, then $|var_{pi}| = 1$.
5. For each variable denoting a behavior, threads associated to this value must be present in the system and share the same marker as the one of the variable.

Proof A.2 *The proof can be made by induction on created terms issued from the initial state $\beta(\mathcal{S}, \emptyset)$. We give here only proof sketches of the different cases:*

1. *Initial threads as well as every created threads are computed using the β function and correspond to either static actors, dynamic ones, or messages. Therefore they are able to exhibit partial interactions as defined in 3.2.4. We recall that we only consider closed term. Then, the initial threads are defined after a sequence of ν operators that bind their variables. By induction on the number of computed transitions, we can show that each transition using one of the two formal rules preserves the property: each variable used in one thread partial interaction is defined and is bound either by an internal ν operator, a ζ operator, a message argument or previously in the matching actor.*
2. *By definition of the abstract syntax extraction in 3.2.4. Each actor on program point ι associated to a syntactic behavior $(\{\iota_i\})$ is mapped, by the β function, to one thread (ι, E) and a set of threads (ι_i, E_i) . Each of those static threads is then be launched by the `launch` primitive. By definition, all the behavior branches are associated to threads that exhibit behavior partial interactions.*
3. *Similar to the previous case. Using the result of 1, the variable s is defined and denotes the program point of the static actor defining the behavior.*
4. *By definition of the syntax extraction.*
5. *All threads representing the different branches of a behavior are launched together. Therefore they are all present and share the same marker.*

To simplify the translation and allow us to differentiate between recursive instances of the same variable declaration (*i.e.* name binder), we define an auxiliary function f which maps each pair $(p, m) \in \mathcal{L} \times \mathcal{M}$ to the name p^m iff p is the label corresponding to a term νa^p and the term $[m_i^{\iota_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$ iff p is a term $a \triangleright^p [m_i^{\iota_i}(\tilde{x}_i) = \zeta(e_i, s_i)C_i^{i=1, \dots, n}]$.

Such a function also allows us to replace dynamically defined actors by their static equivalent ones. The α -conversion rule of the CAP congruence relation allows us to rearrange the term.

Let $\{c_i \mid i \in \llbracket 1; k \rrbracket\}$ defined as

$$\left\{ f(E(x)) \left| \begin{array}{l} (p, id, E) \in C, \\ E(x) \text{ defined, and} \\ fst(E(x)) \text{ a name} \end{array} \right. \right\}$$

be the set of actors' names used in the term.

We define a translation function Π which maps a set of threads denoting a well formed non standard configuration C to the corresponding CAP configuration. The Π function is defined by:

$$\Pi(C) = (\nu c_1) \dots (\nu c_k)(M_1 \parallel \dots \parallel M_p \parallel A_1 \parallel \dots \parallel A_q)$$

We define C as the disjoint union of M , A and B :

$$C = M \cup A_s \cup A_d \cup B$$

where M is the set of threads associated to message_n partial interactions, A_s the set of threads associated to static_actor_n partial interactions, A_d the set of threads associated to dynamic_actor partial interactions, and, finally, B the set of threads associated to behavior_n partial interactions.

The Π function can be recursively applied on the set of threads in C , each iteration computing a message or an actor of the resulting CAP term.

We construct $\{M_i\}$ and $\{A_j\}$ as follows:

- $i \in \llbracket 1; \text{Card}(M) \rrbracket$.
- M_i is the message $\alpha \triangleleft^l(\tilde{x})$ corresponding to the translation of the thread $(l, \text{id}, E) \in M$.
- $j \in \llbracket 1; \text{Card}(A_s \cup A_d) \rrbracket$.
- A_j is the actor $\alpha \triangleright^l [m_i^{l_i}(\tilde{x}_i) = \zeta(e_i, s_i) C_i^{i=1, \dots, n}]$ associated to the thread $(l, \text{id}, E) \in A_s \cup A_d$.
 - when $(l, \text{id}, E) \in A_s$, the actor is obtained from the sub-term associated to the program point l ;
 - when $(l, \text{id}, E) \in A_d$, the actor is composed of all behaviors represented by its associated threads $(l_i, \text{id}, E) \in B$; *i.e.* behavior threads that program points are associated by behavior_set function to the second variable value of the actor thread and that markers are equal to the one of this same variable. More formally, the second variable of the actor thread is associated to the pair (l', id') , $\text{id} = \text{id}'$ and all l_i are associated to l' by the behavior_set function.

As the C term is well formed, when a dynamic actor is in the system, all the behaviors of its behavior set must be in the system too.

Finally, in both M_i and A_j , we update E . We replace α and each free variable of \tilde{x} , respectively α and each free variable in all C_i , with their image by function $f: x \mapsto f(E(x))$.

The translation system is well defined thanks to the congruence rules: associativity, commutativity and swapping.

Correspondence

The following theorem states that CAP standard semantics and its non standard semantics are in strong bisimulation. They share equivalent initial states and each possible set of transitions from the initial state in the non standard semantics (respectively in the standard one) is computable in the standard one (respectively in the non standard one).

Theorem A.3 We have $\mathcal{S} \equiv \Pi(\mathcal{C}_0)$, and for each non standard configuration C and for each word $u \in (\mathcal{L}^2 \cup \mathcal{L}^3)^*$ such that $\mathcal{C}_0 \xrightarrow{u}^* C$:

1. $\forall \lambda \in (\mathcal{L}^2 \cup \mathcal{L}^3), C \xrightarrow{\lambda} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;
2. $\forall \lambda \in (\mathcal{L}^2 \cup \mathcal{L}^3), \Pi(C) \xrightarrow{\lambda} P \implies \exists D, \begin{cases} C \xrightarrow{\lambda} D \\ \Pi(D) \equiv P \end{cases}$

Proof A.4 Let C be a non standard configuration and let u be a word in $(\mathcal{L}^2 \cup \mathcal{L}^3)^*$ such that $\mathcal{C}_0 \xrightarrow{u}^* C$,

1. Let C' be a non standard configuration such that $C \xrightarrow{\lambda} C'$. Suppose that C contains only interacting threads. Our coding contains two transition rules, we check the property for each. In order to simplify the proof, we denote by \mathcal{E} and \mathcal{S} the address and behavior set of the interacting actor and by \mathcal{M} the environment of the interacting message.

- a) **Static-trans rule.** Such a rule defines the interaction between two threads. Necessarily, C must contain:

$$(p_1, id_1, E_1) \quad (p_2, id_2, E_2)$$

where partial interactions associated to program points are:

- p_1 : (static_actor_n, [ego, label], [e_i, s_i, \widetilde{y}_k], $\beta(C_i, \emptyset)$)
- p_2 : (message_n, [dest, label, \widetilde{x}_l], \emptyset)

with $|\widetilde{y}_k| = |\widetilde{x}_l| = n$. Then, we have the following relations $E_1(\text{ego}) = E_2(\text{dest})$ and $E_1(\text{label}) = E_2(\text{label})$. The C' term obtained after transition $\lambda = (p_1, p_2)$ is:

$$\left\{ \begin{array}{l} (p_j, id_j, E_j) \text{ s.t. } (p_j, Es_j) \in \beta(C_i, \emptyset) \text{ and} \\ E_j = Es_j \left[\begin{array}{l} e_i \mapsto E_2(\text{ego}) \\ s_i \mapsto I^1 = (p_1, id_1) \\ \forall k \in \llbracket 1; n \rrbracket, \\ y_k \mapsto E_2(x_k) \end{array} \right] \end{array} \right\}$$

Standard configuration $\Pi(C')$ is the closed term composed of the set of messages and actors with their appropriate behaviors, as defined by the C' term. Let us study each C_i case by induction on CAP syntax. We have here $\mathcal{E} = E_1(\text{ego})$, $\mathcal{S} = I_1$ and $\mathcal{M} = E_2$.

- when $C_i = a \triangleright^{p_a} [m_l^{p_{m_l}}(\widetilde{z}_l) = \zeta(e_l, s_l)C_l]$, then, by definition of β , $\beta(C_i, \emptyset) = (p_a, [\text{self} \mapsto p_a]) \cup \bigcup (p_{m_l}, [])$.

The exhibits function maps each program point p_{m_l} to a partial interaction:

$$(\text{behavior}_n, [m_l], [e_l, s_l, \tilde{z}_l], \beta(C_l, \emptyset))$$

After value passing, we obtain for C' :

$$\left\{ \left(p_a, \text{new_id}, E_a \left[\begin{array}{l} e_i \mapsto \mathcal{E} \\ s_i \mapsto \mathcal{S} \\ \forall k \in \llbracket 1; n \rrbracket, \\ y_k \mapsto \mathcal{M}(x_k) \\ x \mapsto E_1(x) \end{array} \right. \middle| \begin{array}{l} \text{such that } E_a(x) \\ \text{is defined iff} \\ x \in \text{interface}(p_a) \end{array} \right. \right) \right\}$$

$$\cup \cup_l \left\{ \left(p_l, \text{new_id}, E_l \left[\begin{array}{l} e_i \mapsto \mathcal{E} \\ s_i \mapsto \mathcal{S} \\ \forall k \in \llbracket 1; n \rrbracket, \\ y_k \mapsto \mathcal{M}(x_k) \\ x \mapsto E_1(x) \end{array} \right. \middle| \begin{array}{l} \text{such that } E_l(x) \\ \text{is defined iff} \\ x \in \text{interface}(p_l) \end{array} \right. \right) \right\}_l$$

The term translation by Π function gives

$$\nu \tilde{c}, a \triangleright^{p_a} [m_l^{p_l}(\tilde{z}_l) = \zeta(e_l, s_l)C_l]$$

where $\tilde{c} = \{x \mid E_a(x) \text{ where } E_l(x) \text{ is defined and denotes a } \nu \text{ program point}\}$.

- when $C_i = a \triangleright^{p_a} b$, then necessarily, the set of behaviors denoted by the b variables has already been defined and is in the current configuration. We recall that we only consider closed terms. Therefore b is defined and denotes a set of behaviors of the past. The resulting term is similar to the preceding case without introducing behavior threads on program points p_i .
- when $C_i = a \triangleleft^{p_a} m(\tilde{z})$, then by definition of β , $\beta(C_i, \emptyset) = (p_a, \emptyset)$. The exhibits function gives:

$$(\text{message}_n, [a, m, \tilde{z}], [], \emptyset)$$

The p_a interface is $\{a\} \cup \mathcal{FN}(\tilde{z})$. The only thread we obtain, in the resulting term C' , once the value passing is computed, is:

$$\left\{ \left(p_a, \text{new_id}, E \left[\begin{array}{l} e_i \mapsto \mathcal{E} \\ s_i \mapsto \mathcal{S} \\ \forall k \in \llbracket 1; n \rrbracket, \\ y_k \mapsto \mathcal{M}(x_k) \end{array} \right. \middle| \begin{array}{l} \text{such that } E(x) \\ \text{is defined iff} \\ x \in \text{interface}(p_a) \end{array} \right. \right) \right\}$$

Its translation by Π is:

$$\nu \tilde{c}, a \triangleleft^{p_a} m(\tilde{z})$$

where $\tilde{c} = \{x \mid E(x) \text{ is defined and denotes a } \nu \text{ program point}\}$.

b) **Dynamic-trans rule.** Such a rule defines the interaction between three threads. Necessarily C must contain:

$$(p_1, id_1, E_1) \quad (p_2, id_2, E_2) \quad (p_3, id_3, E_3)$$

where partial interactions associated to program points are:

- p_1 : (behavior_n, [label], [e_i, s_i, \tilde{y}_k], $\beta(C_i, \emptyset)$)
- p_2 : (actor, [ego, self], [], \emptyset)
- p_3 : (message_n, [dest, label, \tilde{x}_l], \emptyset)

with $|\tilde{y}_k| = |\tilde{x}_l| = n$. Then, we have the following relations

$$\left\{ \begin{array}{l} E_2(\text{ego}) = E_3(\text{dest}), \\ E_1(\text{label}) = E_3(\text{label}) \text{ and} \\ \text{behavior_set}((p_1, id_1)) = E_2(\text{self}). \end{array} \right.$$

The C' term obtained after transition $\lambda = (p_1, p_2, p_3)$ is:

$$\left\{ \begin{array}{l} \{(p_1, id_1, E_1)\} \cup \\ (p_j, \text{new_id}, E_j) \end{array} \middle| \begin{array}{l} (p_j, E_{s_j}) \in \beta(C_i, \emptyset) \\ E_j = E_{s_j} \left[\begin{array}{l} e_i \mapsto E_2(\text{ego}) \\ s_i \mapsto E_2(\text{self}) \\ \forall k \in \llbracket 1; n \rrbracket, \\ y_k \mapsto E_3(x_k) \end{array} \right] \end{array} \right\}$$

where $\text{new_id} = id_2.p_2$.

Standard configuration $\Pi(C')$ is, as preceding, the closed term composed of the set of messages and actors with their appropriate behaviors, as defined by the C' term. We have now to study each C_i case by induction on CAP syntax. We obtain the same results as in the static_rule case with $\mathcal{E} = E_2(\text{ego})$, $\mathcal{S} = E_2(\text{self})$ and $\mathcal{M} = E_3$.

c) Finally, the last cases are managed by an induction over the shape of C_i terms, independently of the matching rule.

- when $C_i = \nu a^\alpha, C_a$, then $\beta(C_i, \emptyset) = \beta(C_a, [a \mapsto \alpha])$. By induction hypothesis, the resulting term of the update of the static environment of program points $\beta(C_a, \emptyset)$ is C_a . The variable a may be free in the C_a term.

The resulting term of the environment update with the relation $a \mapsto \alpha, \text{new_id}$, is translated by Π to $\nu a^\alpha C_a$;

- when $C_i = C_1 \parallel C_2$ then $\beta(C_i, \emptyset) = \beta(C_1, \emptyset) \cup \beta(C_2, \emptyset)$. By induction, we have $\beta(C_1, \emptyset)$ which is translated in C_1 by Π and $\beta(C_2, \emptyset)$ which is translated in C_2 . The term $\beta(C_i, \emptyset)$ translation with updated environment by value passing is the closed term of CAP composed of both actors and messages of C_1 and C_2 ;
- when $C_i = \emptyset$ then $\beta(C_i, \emptyset) = \emptyset$. It's a trivial implication.

We have shown that the implication is valid if term C contains only matching threads. Compatibility rules as well as congruence relations allow us to compute a transition step when the actor can handle more behaviors and when interacting threads are under a variable restriction or in parallel with other threads. So, the implication is true in the general case for well formed configurations.

We have the implication $\forall \lambda \in (\mathcal{L}^2 \cup \mathcal{L}^3), C \xrightarrow{\lambda} C' \implies \Pi(C) \xrightarrow{\lambda} \Pi(C')$;

2. Let P be a standard configuration such that $\Pi(C) \xrightarrow{\lambda} P$. Let $(p_i)_i = \lambda$ the transition label. Configuration $\Pi(C)$ contains at least the interacting threads $\{p_i\}$. By definition of translation function Π , non standard term C must contain at least some threads (p_1, id_1, E_1) , (p_2, id_2, E_2) and (p_3, id_3, E_3) or (p_1, id_1, E_1) and (p_2, id_2, E_2) , depending on the matching rule, with the appropriate constraints on markers and environments to allow transition λ . Therefore, there exists a non standard configuration D image of C by transition λ . Let $\Pi(D)$ be its image in the CAP standard syntax by the translation function Π . By reusing preceding property, and the fact that term C' is well formed, we obtain $\Pi(D) \equiv P$ by α -conversion and extrusion.

Therefore we have $\Pi(C) \xrightarrow{\lambda} P \implies \exists D, C \xrightarrow{\lambda} D$ and $\Pi(D) \equiv P$.

3. Let us show that $\mathcal{S} \equiv \Pi(\mathcal{C}_0)$. By induction, we show that $\forall C_i$ closed, $\beta(C_i, \emptyset)$ is closed. Furthermore, $\Pi(\beta(C_i, \emptyset)) \equiv C_i$ is closed. Let $C_0 = \mathcal{C}_0$ be the initial configuration. As $\mathcal{C}_0 = \beta(\mathcal{S}, \emptyset)$ by definition, we have $\Pi(C_0) \equiv \mathcal{S}$.

A.2 PARTITIONED ABSTRACT DOMAIN

Lemma A.5 (monotonicity of continuation over-approximation) *The function `conts_of_unit` is a sound over-approximation of the partitioning of a continuation among its different target address. Furthermore this function is monotonic.*

Proof A.6 *Let us consider a continuation `cont`. Let $(a, id) \in \mathcal{L}_v \times \mathcal{M}$ be an address. Let us consider the concrete partitioning of the launched continuation `launch(cont)`. We denote by `cont| (a, id)` the subset of `cont` that is used to create threads on address (a, id) .*

We prove that the concrete subset `cont| (a, id)` is computed by the function `conts_of_unit` and associated to the binder a .

According to the definition of `conts_of_units(cont, cf, mol)`, then necessarily

- $\exists(\alpha, \text{cont}_\alpha) \in \text{conts_of_unit_by_binder}(\text{cont}, \text{cf})$ and
- $\text{cont}_{|(\alpha, \text{id})} \in \text{conts_by_marker}(\text{cont}_\alpha, \text{mol})$.

The continuation subset $\text{cont}_{|(\alpha, \text{id})}$ is composed of threads which addresses are equal to (α, id) . Let $X \in \wp(\mathcal{V})$ be the set of variables in cont that are bound to value (α, id) . Necessarily, all threads $(p, E_S) \in \text{cont}_{|(\alpha, \text{id})}$ are such that $\text{address_var}(p) \in X$.

By soundness of the control flow abstraction, these variables in X are associated to a sound over-approximation of there binder. Then $\text{conts_of_unit_by_binder}(\text{cont}, \text{cf})$ returns a pair $(\alpha, \text{cont}_\alpha)$ such that $\text{cont} \subseteq \text{cont}_\alpha$.

We now have to prove the $\text{cont}_{|(\alpha, \text{id})} \in \text{conts_by_marker}(\text{cont}_\alpha, \text{mol})$. Let us partitioned the set cont by the address variables in X . We have $\text{cont} = c_1 \cup c_2 \cup \dots \cup c_n$.

Again by soundness of the control flow abstraction, we have a sound over-approximation of equality and disequality relations in mol between variables in X .

By definition of conts_by_marker , we can build cont using equivalence classes in $\text{cont}_{\alpha \sim_{\text{mol}}}$.

We have $(\alpha, \text{cont}_{|(\alpha, \text{id})}) \in \text{conts_of_units}(\text{cont}, \text{cf}, \text{mol})$.

Concerning the monotonicity of conts_of_unit , the first part returns greater continuations with a greater control flow abstraction. Then the greater the abstract element is in the second part, the less information is available and the more possible combination of variables computed. Existing continuations are preserved. The function is monotonic.

Theorem A.7 (4.20) The abstraction $(\mathcal{C}^\#, \sqsubseteq^\#, \sqcup^\#, \perp^\#, \gamma^\#, \mathcal{C}_0^\#, \rightarrow^\#, \nabla^\#)$ is a sound abstraction with respect to the definition 3.3 considering a sound underlying control flow abstraction and an abstract domain $\mathcal{C}^{\text{base}}$ satisfying the soundness assumption of definition 4.7.

Proof A.8 (4.21)

1. Properties (1)(2)(3) requiring a pre-order, a join operator, a bottom element are satisfied by our definitions:

- the pre-order $\sqsubseteq^\#$ is defined as

$$(cf_1, (\text{beh}_1, \text{cu}_1)) \sqsubseteq^\# (cf_2, (\text{beh}_2, \text{cu}_2)) \Leftrightarrow \begin{cases} cf_1 \sqsubseteq^{\text{env}} cf_2 \wedge \\ \text{beh}_1 \sqsubseteq^{\text{base}} \text{beh}_2 \wedge \\ \text{cu}_1 \sqsubseteq^{\text{base}} \text{cu}_2 \end{cases}$$

where $\sqsubseteq^{\text{base}}$ is the point-wise extension of $\sqsubseteq^{\text{base}}$ on $\text{Unit} \rightarrow \mathcal{C}^{\text{base}}$.

- then the component-wise defined join $\sqcup^\#$ and bottom $\perp^\#$ satisfy prop. (2) and prop. (3), respectively.

2. Prop. (4) holds. The widening operator is defined component-wise. Each component-wise widening satisfies the property on its associated lattice element. Similarly to the join operator, the property applies on the component-wise defined widening.
3. Both concretization functions γ_{env} and γ_{base} of underlying abstract domains satisfy the prop. (5) on their associated domain.

We recall the $\gamma^\#$ definition:

$$\gamma^\#(\text{cf_flow}, \text{part}) \triangleq \gamma_{\text{env}}(\text{cf_flow}) \cap \gamma_{\text{part}}(\text{part})$$

It is then monotonic iff the γ_{part} function is monotonic. We recall its definition:

$$\left\{ \begin{array}{l} (u, C) \left| \begin{array}{l} \forall (p, \text{id}, E) \in C, \\ \text{either } p \in \mathcal{L}_b \text{ and } (p, \text{id}, E) \in \gamma_{\text{base}}(\text{beh}) \\ \text{either } \left\{ \begin{array}{l} p \in \mathcal{L}_a \cup \mathcal{L}_m, \\ \exists \text{id}' \in \mathcal{M}, E[\text{address_var}(p)] = (a, \text{id}'), \\ \exists (a, \text{unit}_a) \in \text{cu}, (p, \text{id}, E) \in \gamma_{\text{base}}(\text{unit}_a) \end{array} \right. \end{array} \right. \right. \\ \left. \right\} \subseteq \gamma_{\text{part}}(\text{beh}, \text{cu})$$

This concretization function for the partitioned part of the abstract element can be seen as the union of the concretization, using γ_{base} of the partition unit. It is therefore also monotonic.

4. We now prove that the prop. (6) is satisfied. We have to show that considering:

- two concrete configurations $(u, C), (u', C') \in \Sigma^* \times \mathcal{C}$;
- a abstract element $C^\# \in \mathcal{C}^\#$;
- such that $(u, C) \in \gamma^\#(C^\#)$ and
- $\exists \lambda \in \mathcal{L}_p^{\{2\} \cup \{3\}}$ such that $u' = \lambda.u$ and $(u, C) \xrightarrow{\lambda} (u', C')$

then

$$\exists C'^\# \in \mathcal{C}^\# \text{ such that } C^\# \xrightarrow{\lambda} C'^\# \text{ and } (u', C') \in \gamma^\#(C'^\#).$$

We know that $(u, C) \in \gamma^\#(C^\#)$ and $(u, C) \xrightarrow{\lambda} (u', C')$. Let $C^\# = (\text{cf}, \text{part})$. Then following the definition of $\gamma^\#$, we have $(u, C) \in \gamma_{\text{env}}(\text{cf})$. Furthermore, using the hypothesis of \mathcal{C}^{env} as a sound control flow abstraction, we have $\exists \text{cf} \in \mathcal{C}^\#, \text{cf} \xrightarrow{\lambda(p^k)_k}_{\text{env}} \text{cf}'$ and $(u', C') \in \gamma_{\text{env}}(\text{cf}')$.

We now have to prove that the partitioned part allows the abstract transition computation and that (u', C') is in the concretization of the resulting abstract partitioned element.

Let $(\mathbf{a}, \mathbf{id}) \in \mathcal{L}_v \times \mathcal{M}$ be the address on which the transition λ occurs in C . The concrete configuration C contains at least two threads $(p_a, \mathbf{id}_a, E_a)$ and $(p_m, \mathbf{id}_m, E_m)$ associated to this address $(\mathbf{a}, \mathbf{id})$ and interacting by λ .

Then according to the definition of γ_{part} , it exists $(\mathbf{a}, \text{unit}_a) \in \text{cu}$ where $\text{part} = (\text{beh}, \text{cu})$ such that

$$\{(p_a, \mathbf{id}_a, E_a), (p_m, \mathbf{id}_m, E_m)\} \subseteq \gamma_{\text{base}}(\text{unit}_a)$$

Let us consider the two different cases depending on the matched rule: involving a syntactically defined actor or a dynamic one.

- **Static-trans rule.** In that case, according to the soundness assumption of the $\text{sync}^{\text{base}}$ primitive:

$$\left\{ \begin{array}{l} (\mathbf{u}, C) \in \gamma_{\text{base}}(\mathbf{x}) \\ \forall k, (p^k, \mathbf{id}^k, E^k) \in C \text{ and} \\ \exists C' \text{ s.t. } C \xrightarrow{\mathcal{R}, (p^k)} C' \end{array} \right\} \\ \subseteq \gamma_{\text{base}}(\text{sync}^{\text{base}}(\mathcal{R}, (p^k), (\text{parameter}^k), \mathbf{x}))$$

We have at least

$$\{(\mathbf{u}, \{(p_a, \mathbf{id}_a, E_a), (p_m, \mathbf{id}_m, E_m)\})\} \\ \in \gamma_{\text{base}}(\text{sync}^{\text{base}}(\text{static_trans}, \lambda, (\text{parameter}^k), \text{unit}_a))$$

Then since γ_{base} is strict, we have

$$\text{sync}^{\text{base}}(\text{static_trans}, \lambda, (\text{parameter}^k), \text{unit}_a) \neq \perp^{\text{base}}.$$

- **Dynamic-trans rule.** A similar reasoning occurs here. The only difference is that we consider $\text{unit}_a \sqcup^{\text{base}} \text{beh}$ instead of unit_a . We have $(p_b, \mathbf{id}_b, E_b)$ the behavior thread which is present in the concretization of the partitioned part, furthermore in $\gamma_{\text{base}}(\text{beh})$. The result is identical: using the soundness assumption of $\text{sync}^{\text{base}}$ and by strictness of γ^{base} , $\text{sync}^{\text{base}}(\text{static_trans}, \lambda, (\text{parameter}^k), \text{unit}_a \sqcup^{\text{base}} \text{beh}) \neq \perp^{\text{base}}$.

In both case, we have at least a partition unit associated to the address \mathbf{a} which has a non bottom image by $\text{sync}^{\text{base}}$. Let $\text{synced}^{\text{base}}$ be such value. The $\text{sync}^{\text{part}}$ primitive gives then a non empty set of synchronized units. The abstract transition can occur.

Let us now build such abstract image $(\text{beh}', \text{cu}') \in \text{Part}(\text{Units}, \mathcal{C}^{\text{base}})$ and show that $(\mathbf{u}', C') \in \gamma^{\text{part}}((\text{beh}', \text{cu}'))$.

Let us partition the resulting concrete configuration C' according to thread address. Let us prove that for each partition $C'_{|(b, \mathbf{id}')}$ denoting the threads associated to the address $(\mathbf{b}, \mathbf{id}') \in \mathcal{L}_v \times \mathcal{M}$ we have $(\mathbf{u}', C'_{|(b, \mathbf{id}')}) \subseteq \gamma^{\text{base}}(\text{unit}'_b)$

where unit_b and unit'_b denote the elements of $\mathcal{C}^{\text{base}}$, when they exist, associated to the binder b in cu and cu' , respectively.

We also have to prove that the behavior threads of C' are in $\gamma_{\text{base}}(\text{beh}')$.

There are four cases for partitions:

- a) $C'_{|(b, \text{id}')}$ denotes the interacting unit, $(b, \text{id}') = (a, \text{id})$;
 - b) $C'_{|(b, \text{id}')}$ results from the launching of new threads in $C_{|(b, \text{id}')}$;
 - c) $C'_{|(b, \text{id}')}$ denotes the creation of a new unit since $C_{|(b, \text{id}')} = \emptyset$;
 - d) $C'_{|(b, \text{id}')} = C_{|(b, \text{id}')}$ is an untouched partition by the computed transition.
- a) In the first case, $C'_{|(a, \text{id})}$ is defined as

$$C_{|(a, \text{id})} \setminus \{(p_a, \text{id}_a, E_a), (p_m, \text{id}_m, E_m)\} \cup \text{launched_threads}$$

where launched_threads is defined as the threads built from the sub-set cont of static threads continuation associated to address (a, id) .

Let $(a, \text{cont}_a) \in \overline{\text{conts_of_units}}$. Relying on the lemma A.5, we have $\text{cont} \in \text{cont}_a$. Then by monotonicity of γ^{base} and by definition of launched_units , we need to prove that

$$(\mathbf{u}, C'_{|(a, \text{id})}) \in \gamma^{\text{base}}(\text{update}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\text{base}}, a, \text{unit}_a, \text{cont}))$$

Using the soundness assumption of $\text{update}^{\text{base}}$ primitive, we have:

$$\left((\mathbf{u}', C') \mid \exists (\mathbf{u}, C) \text{ s.t. } \left\{ \begin{array}{l} \exists \text{id} \in \mathcal{M}, (\mathbf{u}, C_{|(a, \text{id})}) \subseteq \gamma_{\text{base}}(\text{unit}_a) \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|(a, \text{id})} \cap \gamma_{\text{base}}(\text{synced}^{\text{base}}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ C'_{|(a, \text{id})} = C_{|(a, \text{id})} \setminus \left\{ \begin{array}{l} (p^a, \text{id}^a, E^a); \\ (p^m, \text{id}^m, E^m) \end{array} \right\} \cup \text{launch}(\text{cont}) \end{array} \right. \right) \right. \\ \subseteq \gamma_{\text{base}}(\text{update}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\text{base}}, a, \text{unit}_a, \text{cont}))$$

Then $(\mathbf{u}', C'_{|(a, \text{id})}) \in \gamma^{\text{base}}(\text{unit}'_a)$.

- b) In the second case, $C'_{|(b, \text{id}')}$ is defined as $C_{|(b, \text{id}')} \cup \text{launched_threads}$ where launched_threads is defined as the threads built from the sub-set cont of static threads continuation associated to address (b, id') .

Similarly to the preceding case, using the lemma A.5, $\exists (b, \text{cont}_b) \in \overline{\text{conts_of_unit}}$ and $\text{cont} \in \text{cont}_b$.

Then by monotonicity of γ^{base} and by definition of launched_units , we need to prove that

$$(\mathbf{u}, C'_{|(b, \text{id}')}) \in \gamma^{\text{base}}(\text{launch}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\text{base}}, b, \text{unit}_b, \text{cont})).$$

The soundness assumption of $\text{launch}^{\text{base}}$ gives:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_v \times \mathcal{M}, \text{ s.t.} \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|x} \cap \gamma_{\text{base}}(\text{synced}^{\text{base}}) \\ \forall \text{id} \in \mathcal{M}, (u, C_{|(b, \text{id})}) \subseteq \gamma_{\text{base}}(\text{unit}_b) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ \exists \text{id}' \in \mathcal{M}, C'_{|(b, \text{id}')} = C_{|(b, \text{id}')} \cup \text{launch}(\text{cont}) \end{array} \right. \right\} \\ \subseteq \gamma_{\text{base}}(\text{launch}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\text{base}}, b, \text{unit}_b, \text{cont})).$$

Then $(u', C'_{|(b, \text{id}')}) \in \gamma^{\text{base}}(\text{unit}'_b)$.

c) In the third case, $C'_{|(b, \text{id}')}$ is defined as `launched_threads`, the threads built from the sub-set `cont` of static threads continuation associated to newly created address (b, id') . The set `cont` can be exactly identified in (pi^k) since there is a unique occurrence of a ν operator on program point b in the associated continuations.

Again, by monotonicity of γ^{base} and by definition of `launched_units`, we need to prove that

$$(u, C'_{|(b, \text{id}')}) \in \gamma^{\text{base}}(\text{launch}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), b, \text{synced}^{\text{base}}, \text{cont})).$$

The soundness assumption of $\text{launch}^{\text{base}}$ is recalled above. In case, of a \perp_{Base} value for unit_b , we have, for all $\text{id} \in \mathcal{M}$, the configurations $C_{|(b, \text{id})} = \emptyset$ and $C'_{|(b, \text{id}')} = \text{launch}(\text{cont})$.

Then $(u', C'_{|(b, \text{id}')}) \in \gamma^{\text{base}}(\text{unit}'_b)$.

d) In case of untouched partition unit, the property is satisfied since cu' is defined as $cu \sqcup \text{launched_units}$ then $cu \sqsubseteq^{\text{base}} cu'$. By monotonicity of γ^{base} we have the property.

A similar mechanism apply for beh' , it is built using `launch_beh`^{base} and by soundness assumption we have:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_v \times \mathcal{M}, \text{ s.t.} \\ \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \subseteq C_{|x} \cap \gamma_{\text{base}}(\text{synced}) \\ C_{|\text{beh}} \subseteq \gamma_{\text{base}}(\text{beh}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ C'_{|\text{beh}} = C_{|\text{beh}} \cup \text{launch}(\text{cont}_{\text{beh}}) \end{array} \right. \right\} \\ \subseteq \gamma_{\text{base}}(\text{launch_beh}^{\text{base}}(\mathcal{R}, (p^k), (pi^k), \text{synced}, \text{beh}, \text{cont}_{\text{beh}}))$$

Then we have the set of behavior threads in C' included in $\gamma^{\text{base}}(\text{beh}')$.

Finally all units are sound, and by construction of γ^{part} ,

$$(u', C') \in \gamma^{\text{part}}((\text{beh}', cu')).$$

5. Finally the last property (7) to be proved is the initial element abstraction: $C_0 \subseteq \gamma^\#(C_0^\#)$.

Let init_s be the initial set of static threads. init_s can be partitioned according to threads address. We obtain init_beh the set of initial thread associated to program points in \mathcal{L}_b and init_u the set of initial thread associated to the address binder on program point u .

By soundness of the underlying control flow abstraction, $C_0 \subseteq \gamma^{\text{env}}(C_0^{\text{env}}(\text{init}_s))$, with $C_0^\# = (C_0^{\text{env}}(\text{init}_s), (C_0^{\text{base}}(\text{init_beh}), C_0^{\text{part}}))$. Similarly by soundness of the underlying abstract domain $\mathcal{C}^{\text{base}}$ and by definition of γ^{part} , $C_{0|(u,\epsilon)} \in \gamma^{\text{base}}(C_0^{\text{base}}(\text{init}_u))$. The same applies for init_beh .

A.3 OCCURRENCE COUNTING WITH TRANSITIONS

Theorem A.9 (5.6) *The abstraction defined by $(\text{Dep} \times \mathbb{N}, \sqsubseteq, \sqcup, \perp, \gamma, C_0^\#, \rightarrow_\#, \nabla)$ is a sound abstraction with respect to definition 3.3.*

Proof A.10 (5.7)

1. Properties (1)(2)(3) requiring a pre-order, a join operator, a bottom element are satisfied by our definitions;
2. Prop. (4) holds. The widening operator relies on the widening operator defined in the underlying numerical abstract domain. The dependency part is built on finite set and do not necessitate widening;
3. Our gamma function is monotonic and satisfy prop. (5). It is build on the gamma function of the numerical abstraction as well as the dependency abstraction. The first one satisfies the property by construction. In the second, adding elements to the set of dependencies allows more configurations to be represented as well as all their prefixes;
4. The prop. (6) is satisfied. We consider a concrete configuration $(u, C) \in \Sigma^* \times \mathcal{C}$ and an abstract element $C^\# \in \text{Dep} \times \mathbb{N}$ such that
 - $(u, C) \in \gamma(C^\#)$;
 - $\exists \lambda \in \mathcal{L}^{\{2\} \cup \{3\}}, C' \in \mathcal{C}$, such that $C \xrightarrow{\lambda} C'$.

We have now to show that $C'^\#$ exists such that $C \xrightarrow{\lambda} C'^\#$ then that $(u, \lambda, C') \in \gamma(C'^\#)$.

According to the operational semantics definition, $C'^\#$ exists iff the SYNC primitive returns a non bottom value. The occurrence counting part of $C'^\#$ soundly approximates the occurrence of threads and transition labels in C .

The modified synchronization ensures that not only interacting threads (p, T) are present but also that dependent transitions (p, F) occur.

In the first case, using the soundness property of the underlying occurrence counting abstraction, all interacting program point variables (p, T) are present in concretizations of the occurrence counting part of $C^\#$.

We now show that since $(u, C) \in \gamma(C^\#)$ then each dependent transition labeled by p is in u and therefore the variable $(p, F) \in \mathcal{V}_c$ is associated to a positive occurrence in $C^\#$.

For each interacting thread (p^k, id^k, E^k) in the λ labeled transition, let u^k be the scattered sub-word of u such that it describes the transitions leading to the creation of the thread.

Then, since each of these threads is in C and $(u, C) \in \gamma(C^\#)$, we have, for each of these u^k , an associated set of dependencies in $dep^\#$ that describes static transitions in u^k .

Let $p_1^k \rightsquigarrow p_2^k \rightsquigarrow \dots \rightsquigarrow p_n^k \rightsquigarrow p^k$ be the longest right factor of u^k in $dep^\#$ to the interacting thread $(p^k, id^k, E^k) \in C$.

Then according to γ definition, $\forall i \in [1 \dots n], p_i^k \in u^k$ and therefore $p_i^k \in u$.

Since $(u, C) \in \gamma(C^\#)$, (u, C) is also present in the concretization of the occurrence counting part and $\forall k \forall i, p_i^k \in u$ and each variable $(p_i^k) \in \mathcal{V}_c$ has a strictly positive occurrence in the abstract element.

The synchronization gives a non bottom value since $(u, C) \in \gamma(\text{SYNC}(t, (d, o)))$.

We now show that $(u', C') \in \gamma(C'^\#)$. C' is defined as $C \cup \text{new_threads} \setminus \text{consumed}$ in the concrete semantics.

DYNAMIC TRANSITION In the case where λ denotes a dynamic transition, i.e. with a type replication associated to the behavior branch thread, then the `launch_dep` primitive leaves the dependencies abstract element untouched.

Let us consider each new launched thread, we have two cases, either there is already an existing dependence for this new thread or not.

In the first case, the dependence (p, p') expresses via the γ function that there is a past transition involving an static actor on program point p . This past actor is necessary an ancestor of the current thread since one of its behavior branch is used to launch the thread on program point p' . So, even if this dependence was built by another transition, it is still valid. Since the new launched thread is produced within a dynamic transition, it needs a transition using this past actor on p in u . And we have $(u', C') \in \gamma(C'^\#) = \gamma(C^\#)$.

In the second case, the γ function does not restrict reachable configurations and $(u', C') \in \gamma(C'^\#) = \gamma(C^\#)$.

STATIC TRANSITION In the case where λ denotes a static transition, the primitive `launch_dep` introduces a new constraint (λ, p) for each thread on p launched.

Old previous threads in C' were present in $\gamma(C^\#)$ and are present in $\gamma(C'^\#)$ by monotonicity. New launched ones also exist since $(\lambda, p) \in \text{dep}^\#$ and $C \xrightarrow{\lambda} C'$. We have $(u.\lambda, C') \in \gamma(C'^\#)$.

5. Finally the last property (7) to be proved is the initial element abstraction: the initial abstract element $C_0^\# \in \mathcal{D}\text{ep} \times \mathbb{N}$ must be such that $(\epsilon, C_0) \subseteq \gamma(C_0^\#)$. The concretization of the empty dependency abstraction element gives only the single element (ϵ, C_0) , no transitions are yet possibly computed. The concretization of the numerical part is already sound and satisfies this property. Their intersection also satisfies it.

A.4 LINEARITY

Theorem A.11 (6.16) *The abstraction defined by $(\mathcal{C}^{\text{lin}'}, \sqsubseteq^{\text{lin}'}, \sqcup^{\text{lin}'}, \perp^{\text{lin}'}, \gamma^{\text{lin}'}, C_0^{\text{lin}'}, \rightarrow_{\text{lin}'}, \nabla^{\text{lin}'})$ is a sound abstraction with respect to definition 3.3.*

Proof A.12 (6.17) 1. Properties (1)(2)(3) requiring a pre-order, a join operator, a bottom element are satisfied by our definitions;

2. Prop. (4) holds. The widening operator is defined as the union one, considering our \mathbb{N}^{flat} usage mode domain of finite depth;
3. Our gamma function is monotonic and satisfy prop. (5). It is defined as the intersection of monotonic maps;
4. The prop. (6) is satisfied.

Let (u, C) be a concrete configuration and let $C^\#$ be an abstract configuration, such that (u, C) is in the concretization $\gamma_{\text{Lin}}(C^\#)$.

Let λ be a transition label and C' be another configuration such that $C \xrightarrow{\lambda} C'$, we must construct $C'^\#$ such that $(u.\lambda, C') \in \gamma(C'^\#)$ and $C^\# \xrightarrow{\lambda}_{\text{Lin}} C'^\#$.

The abstract transition can occur since nothing constrains it in the abstract operational semantics. Let us show now that $(u.\lambda, C') \in \gamma(C'^\#)$.

Let us prove the property for a given address; let say the address $(b, \text{id}) \in \mathcal{L}_b \times \mathcal{M}$. We now consider the different cases: depending whether C is linear for (b, id) and whether the resulting C' is linear for the same address.

1. When C is non linear (at least two actors on (b, id)) then as $(u, C) \in \gamma(C^\#)$, we have $C^\# = (\text{binder}, \text{pps}, \text{dep})$ where $\text{binder}(b) = \top^b$.

By construction of $C'^\#$, in particular the forward flow computation, the launched threads are associated to a non bottom value and their environment variables are linked to their possible binders through the update dependencies relation.

By definition of γ , the concretization part devoted to binders does not constrain at all the configurations and the threads in C' are included in the

concretization of the other part devoted to abstract thread environment and dependencies.

2. When C is linear for (b, id) with one actor on it. Let (p, id', E) be such an actor thread with $add \in I(p)$ its address variable and $E(add) = (b, id)$.

a. either C' is non linear (two actors on (b, id) after the transition)

- a.1) if a single actor is launched. Then the transition did not involve the existing actor on (b, id) . When computing the element $C'^{\#}$, the computed forward flow contains a \bullet mode associated to a variable linked to the binder b through dependencies.

Since the transition did not involve the existing actor, the transitions path necessary to create this previous actor is not a prefix of the path necessary to create this new one. Similarly, considering not anymore transition labels but variable dependencies dep along these paths give two paths in dependencies from the binder definition to its use in actor definitions.

The backward computation, with its fixed point definition, propagates the new \bullet mode along the path until reaching the first node that is in the two paths. Such a node is associated to a continuation where two different threads are associated to the same variable and the same mode \bullet . The mode computation gives the \top^b value and propagates it to the binder.

Like in preceding cases, the \top^b value in $C'^{\#}$ allows any kind of concrete configuration and $(u', C') \in \gamma(C'^{\#})$.

- a.2) if more than one actor is launched on the same address. The forward flow computation detects it if more than one actor are associated to the same variable. In that case, the \top^b value is propagated to the binder mode and $(u', C') \in \gamma(C'^{\#})$.

If all launched actors for the address (b, id) are associated to different variables, a same reasoning as in (a.1) applies: there exists two paths in the dependencies such that a common continuation contains the equality relation between these different variables. We also have $(u', C') \in \gamma(C'^{\#})$.

b. either C' is linear

- b.1) $C'^{\#}$ is such that there is a \top^b value computed for the binder b . Similarly to the non linear case (case 1), any reachable thread (with non bottom value in environment) can be in the concretized configurations.

We have $(u', C') \in \gamma(C'^{\#})$.

- b.2) In the other case, the backward flow with its least fixed point computation did not reach the \top^b value. Any linear configuration for addresses associated to binder b are in the concretization if they

contain only threads with non bottom environment, which is the case for threads in C' .

We have $(u', C') \in \gamma(C^\#)$.

3. When C is linear for the address (b, id) but with no actor on this address.
 - a. either C' is non linear for (b, id) . Therefore there has been a launching during the λ transition of two actors on this same address (b, id) . This case is covered by the case (2.a.2) when these two actors are bound to the same variable address or to different ones but linked to the same possible value.
 - b. either C' is linear for (b, id) . We have exactly the same cases as (b.1) and (b.2)

5. The initial abstraction is defined as the empty set of dependencies, only initial threads are each associated to a non bottom environment and each initial binder is associated to the abstract sum $(+^b)$ of its use in $init_s$ (defining the initial concrete element C_0).

By definition of γ , $\gamma(C_0^{lin'})$ contains all possible finite sets of threads with non bottom environments that satisfy the linearity constraints of their associated binder.

Therefore the concrete element C_0 is in the concretization of $C_0^{lin'}$.

If C_0 is linear for an address (b, id) then the abstract mode associated to b in $C_0^{lin'}$ is \circ or \bullet and only linear configurations for addresses defined on b are allowed. $(\epsilon, C_0) \in \gamma(C_0^{lin'})$.

If C_0 is non linear for an address, the abstract mode associated to its binder is \top^b and its concretization is not constrained for this address and $(\epsilon, C_0) \in \gamma(C_0^{lin'})$.

A.5 ORPHAN MESSAGES

Lemma A.13 (linearity hypothesis) *Let C and C' be two concrete configurations such that $C \rightarrow C'$ with C' containing a new actor on program point p associated to another address a than the one used during the transition from C to C' . Let p_a be the program point of the ancestor of p for the address a . Let $C^\#$ be an abstract configuration such that $C \in \gamma(C^\#)$. Under the linearity hypothesis, the mailbox of the launched actor on p is the one of its ancestor augmented with messages launched during the transition.*

Proof A.14 *Under the linearity hypothesis, there is at most one actor per address in configurations. Associating an actor to a non interacting address (non interacting unit), let say a , requires, for the actor launching the new actor on a , that there is no other actor on this other address. This guaranty can only be obtained by a non empty sequence of transitions initiated by a message sent by an actor on a consumed without re-associating its address to another actor. This last actor on a is the concrete ancestor*

of the new actor on α . Therefore, all messages available to the new actor on α are in the abstract mailbox associated to the node (p_α, False) or in the messages launched during the last transition.

Lemma A.15 (ancestor soundness) *We define by ancestor, the last actor on a given address that has been consumed to generate, may be later, the current launched threads on the same address (unit).*

The monotonic primitive ancestors soundly over-approximates the ancestor of threads.

Proof A.16 *Let us show that the ancestor of the launched threads in the unit is contained in the set of ancestors computed in the abstract.*

The interface abstraction combined with control flow information both allow to soundly over-approximate this relation. A sound set of possible ancestors is considered using the \top automaton approximating the marker of the interacting threads. All program points occurring in such markers are potential ancestors of the current threads in the current unit.

By monotonicity of the construction of the interface and of the marker tree approximation, the ancestor computation is also monotonic. The only best concrete ancestor of one thread is in the set of abstract ancestors computed by the ancestor primitive.

Theorem A.17 (7.17) $(\mathcal{C}, \sqsubseteq, \sqcup, \perp, \gamma, C_0, \nabla)$ *is an abstraction with respect to the definition of 4.7.*

Proof A.18 (7.18) 1. *Properties (1)(2)(3) requiring a pre-order, a join operator, a bottom element are satisfied by our definitions;*

2. *Prop. (4) holds. The widening operator relies on the widening operator defined in the underlying numerical abstract domain. The directed multigraph part of the lattice only admits finite ascending chains;*

3. *Our gamma function is monotonic and satisfy prop. (5). Adding a node and/or an edge between nodes describes more possible mailboxes for more possible actors.*

4. *Prop (7) concerning the initial element abstraction holds: the initial abstract element $C_0^\# \in \mathcal{C}^\#$ must be such that $\{\epsilon\} \times \mathcal{C}_0 \subseteq \gamma(C_0^\#)$. For a given unit, the abstract element is then such that if an actor on program point p is present in the unit:*

- *there is an initial node (p, True) if an actor on this program point is present in the initial configuration;*
- *an abstract mailbox is associated to the value ϵ which denotes initial abstract mailboxes. This mailbox is computed using initial messages on the address denoted by the unit.*

The concretization of such an initial abstract element contains the initial concrete non standard configuration for this address binder.

5. We now have to prove the soundness assumption for our abstract operational primitives:

- Let us prove that

$$\left\{ \begin{array}{l} (\mathbf{u}, \mathbf{C}) \in \gamma(\mathbf{x}) \\ \forall k, (\mathbf{p}^k, \mathbf{id}^k, \mathbf{E}^k) \in \mathbf{C} \text{ and} \\ \exists \mathbf{C}' \text{ s.t. } \mathbf{C} \xrightarrow{\mathcal{R}, (\mathbf{p}^k)} \mathbf{C}' \end{array} \right\} \\ \subseteq \gamma(\text{sync}(\mathcal{R}, (\mathbf{p}^k), (\text{parameter}^k), \mathbf{x}))$$

Let us consider the set of configurations $(\mathbf{u}, \mathbf{C}) \in \gamma(\mathbf{x})$. Let us restrict these transitions to the ones containing the threads $\{(\mathbf{p}^k, \mathbf{id}^k, \mathbf{E}^k)\}$.

Let $\mathbf{p}_a \in (\mathbf{p}^k)$ be the actor thread program point.

According to the definition of γ , $(\mathbf{p}_a, \mathbf{T})$ is a node of \mathbf{x} . Let $\text{mailbox}^\#$ be its associated mailbox as defined in γ .

Let us again restrict such a set of configurations to the ones that can compute the transition $\mathbf{C} \xrightarrow{\mathcal{R}, (\mathbf{p}^k)} \mathbf{C}'$. We denote by \mathbf{C}_{sync} this set of configurations that could compute the transition. Then the concrete mailbox associated to the actor in these \mathbf{C} contains the message label \mathbf{m} .

We have $\{\mathbf{m}\} \subseteq \text{mailbox} \subseteq \gamma(\text{mailbox}^\#)$. By soundness assumption of the underlying numerical abstraction for message label occurrences in mailbox, we have

$$(\text{mailbox} \setminus \{\mathbf{m}\}) \in \gamma(\text{mailbox}^\# \text{ --}_{\text{MX}} \mathbf{1}^\#(\{\mathbf{m}\}))$$

By strictness of the γ_{MX} , we have $\text{mailbox}^\# \text{ --}_{\text{MX}} \mathbf{1}^\#(\{\mathbf{m}\}) \neq \perp_{\text{MX}}$ and $\{(\mathbf{u}, \mathbf{C}) \in \mathbf{C}_{\text{sync}}\} \subseteq \gamma(\text{sync}(\mathcal{R}, (\mathbf{p}^k), (\text{parameter}^k), \mathbf{x}))$

The sync primitive satisfies its soundness assumption.

- Let us prove that

$$\left\{ \begin{array}{l} (\mathbf{u}', \mathbf{C}') \\ \exists (\mathbf{u}, \mathbf{C}) \text{ s.t. } \left\{ \begin{array}{l} \exists \text{id} \in \mathcal{M}, (\mathbf{u}, \mathbf{C}_{|(a, \text{id})}) \subseteq \gamma(\text{synced}^\#) \subseteq \gamma(\text{unit}_a) \\ \{(\mathbf{p}^a, \mathbf{id}^a, \mathbf{E}^a); (\mathbf{p}^m, \mathbf{id}^m, \mathbf{E}^m)\} \subseteq \mathbf{C}_{|(a, \text{id})} \\ \mathbf{C} \xrightarrow{\mathcal{R}, (\mathbf{p}^k), (\mathbf{pi}^k)} \mathbf{C}' \\ \mathbf{C}'_{|(a, \text{id})} = \mathbf{C}_{|(a, \text{id})} \setminus \left\{ \begin{array}{l} (\mathbf{p}^a, \mathbf{id}^a, \mathbf{E}^a); \\ (\mathbf{p}^m, \mathbf{id}^m, \mathbf{E}^m) \end{array} \right\} \\ \cup \text{launch}(\text{cont}_a) \end{array} \right\} \end{array} \right\} \\ \subseteq \gamma(\text{update}^{\text{base}}(\mathcal{R}, (\mathbf{p}^k), (\mathbf{pi}^k), \text{synced}^\#, a, \text{unit}_a, \text{cont}_a))$$

We know that $\exists \text{id} \in \mathcal{M}, (\mathbf{u}, \mathbf{C}_{|(a, \text{id})}) \subseteq \gamma(\text{synced}^\#) \subseteq \gamma(\text{unit}_a)$. In the particular case of this domain $\text{synced}^\# = \text{unit}_a$. Furthermore $\{(\mathbf{p}^a, \mathbf{id}^a, \mathbf{E}^a); (\mathbf{p}^m, \mathbf{id}^m, \mathbf{E}^m)\} \subseteq \mathbf{C}_{|(a, \text{id})}$ and $\mathbf{C} \xrightarrow{\mathcal{R}, (\mathbf{p}^k), (\mathbf{pi}^k)} \mathbf{C}'$.

Then according to the definition of γ , the node $(\mathbf{p}^a, \mathbf{T})$ exists in unit_a and the concretization by γ_{MX} of its associated abstract mailbox $\text{mailbox}^\#$, contains the concrete mailbox mailbox defined as $\mathbf{C}_{|(a, \text{id})} \setminus \{(\mathbf{p}^a, \mathbf{id}^a, \mathbf{E}^a)\}$.

Let us define the resulting mailbox $C'_{|(a, id)}$. It is defined as $C_{|(a, id)} \setminus \{(p^a, id^a, E^a)\} \setminus \{(p^m, id^m, E^m)\} \cup \text{launched}$ where launched correspond to the launching of cont_a .

Computing update on the element unit_a with the associated parameters adds a link between either (p^a, T) and (p', T) when a new actor (p', id', E') is launched in cont_a or between (p^a, T) and (p^a, F) else.

In both case the concretization of the resulting abstract element contains the resulting configuration for this address $C'_{|(a, id)}$.

In the case when cont_a contains an actor, γ gives the configurations that contain this new actor associated to the over-approximation of its mailboxes. This mailbox over-approximation consider the new link added in the graph, consuming the label associated to (p^m, id^m, E^m) and producing $\text{new_messages} = \text{cont}_a \setminus (p', id', E')$. Let $\text{new_messages_labels}$ be their associated multiset of labels.

We have $\text{mailbox}' = \text{mailbox} \setminus \{(p^m, id^m, E^m)\} \cup \text{new_messages}$ and

$$\text{mailbox}'^{\#} \sqsupseteq^{\text{MX}} \text{mailbox}^{\#} -^{\#}_{\text{MX}} 1^{\#}_{\text{MX}}(\{m\}) +^{\#}_{\text{MX}} 1^{\#}_{\text{MX}}(\text{new_messages_labels}).$$

The concretization of this abstract mailbox contains the concrete resulting mailbox $C'_{|(a, id)} \setminus (p', id', E')$.

A similar reasoning applies when cont_a does not contain any actor. the update primitive adds a link between (p_a, T) and (p_a, F) . The concrete resulting mailbox is included in the concretization of the resulting abstract element.

- Let us prove that

$$\left\{ (u', C') \left| \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_v \times \mathcal{M}, \text{ s.t. } C_{|x} \subseteq \{(p^a, id^a, E^a); (p^m, id^m, E^m)\} \\ (u, C_{|x}) \subseteq \gamma(\text{synced}^{\#}) \\ \forall id \in \mathcal{M}, C_{|(b, id)} \subseteq \gamma(\text{unit}_b) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ \exists id' \in \mathcal{M}, C'_{|(b, id')} = C_{|(b, id')} \cup \text{launch}(\text{cont}_b) \\ \subseteq \gamma(\text{launch}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\#}, b, \text{unit}_b, \text{cont}_b)). \end{array} \right. \right. \right\}$$

Like in the previous case, there exists a configuration C allowing to compute a transition on an address x . This transition computation adds the threads defined in cont_b to an address (b, id') .

Let us show that the mailbox associated to this address (b, id') in the resulting configuration is included in the concretization of $\text{launch}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\#}, b, \text{unit}_b, \text{cont}_b)$. We know that $C_{|(b, id)} \subseteq \gamma(\text{unit}_b)$. Let us detail the launch primitive computation.

According to the lemma A.15, the best ancestor is soundly over-approximated by the primitive ancestors.

The next step in the proof is similar to the proof steps considered in the interacting unit: one has to prove that considering the exact ancestor, the resulting abstract interface and mailbox of the unit are sound over-approximations of the concrete resulting configuration.

We now prove that launched threads in the unit are in the concretization of the resulting abstract element. The reasoning for the interface abstract domain is straightforward: the real ancestor is over approximated by the ancestor set, each previously present actor remains untouched by monotonicity of our primitives, while new launched actors are added by the launch primitive.

Steps similar to the interacting threads case also occur. We identify the concrete mailbox associated to the newly launched actor on program point p : it is composed of the ancestor mailbox, the multiset of messages sent to the address since the ancestor consumption, as well as the new messages launched during the current transition.

The lemma A.13 ensures that, under the linearity hypothesis, all messages sent to the address are stored at the ancestor level, either in its mailbox or in the abstract mailbox associated to its removal (with a `False` argument). It also allows to only consider dead nodes when adding a new actor.

The definition of the mailbox computation used before only considered the previous actor, its mailbox and the launched messages. In this kind of “outside interacting unit” launching, the abstract mailbox associated to the node (p_a, False) , where p_a is the ancestor program point, gathers all messages sent to the name without an actor instantiation, considering the local abstract mailbox.

Once again we have $\text{mx}'_p \subseteq \gamma_{\text{MX}}(\text{mx}^{\#}_p)$ for a launched actor on p in $C^{\#}$.

- Similarly the soundness assumption for the `new_launch` primitive has to be proved:

$$\left\{ (u', C') \left| \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_v \times \mathcal{M}, \text{ s.t. } C|_x \subseteq \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \\ (u, C|_x) \subseteq \gamma(\text{synced}^{\#}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ \exists \text{id} \in \mathcal{M} \text{ s.t. } C'_{|(b, \text{id})} = \text{launch}(\text{cont}_b) \\ \subseteq \gamma(\text{new_launch}(\mathcal{R}, (p^k), (pi^k), \text{synced}^{\#}, b, \text{cont}_b)) \end{array} \right. \right. \right\}$$

The proof steps are identical to the ones of the `launch` primitive. A launched actor is a new initial node associated to launched messages. In case of launched messages without an actor, they are associated to messages initially available to all initial actors. The concretization of this element contains the concrete configuration case.

- The `launch_beh` primitive is defined as the identity. In this abstract domain the concretization function does not constrain at all the behavior

threads contained in the configurations. Therefore we directly have the following property:

$$\left\{ (u', C') \mid \exists (u, C) \text{ s.t. } \left\{ \begin{array}{l} \exists x \in \mathcal{L}_V \times \mathcal{M}, \text{ s.t. } C_{|x} \subseteq \{(p^a, \text{id}^a, E^a); (p^m, \text{id}^m, E^m)\} \\ (u, C_{|x}) \subseteq \gamma(\text{synced}^\#) \\ C_{|\text{beh}} \subseteq \gamma(\text{beh}) \\ C \xrightarrow{\mathcal{R}, (p^k), (pi^k)} C' \\ C'_{|\text{beh}} = C_{|\text{beh}} \cup \text{launchcont}_{\text{beh}} \\ \subseteq \gamma(\text{launch_beh}(\mathcal{R}, (p^k), (pi^k), \text{synced}^\#, \text{beh}, \text{cont}_{\text{beh}})) \end{array} \right. \right\}$$

REPLICATING EXAMPLE ANALYSIS

B.1 REPLICATION SERVER EXAMPLE

The following example in Example B.1 is an extension of the replicating server presented in Figure 2.9.

While being quite complex, the system is finite and has no complex interleavings. Let us describe the communication protocol before giving the results of the analyzer.

The transitions begin with the actor `client` receiving the message `init`. It initiates then the replication of the server: it sends to it a message `reify` and waits for an acknowledgment to launch the initial query labeled `m`.

The message `reify` is received by the actor `serv`. It suicides and send its own address and behavior to the duplicating server. The duplicating server installs all the machinery that we described in the first chapter in Figure 2.10, page 29, while replicating and sending the acknowledgment to the client actor.

The message `m` is sent by the client actor to the frontend installed during the replication. This frontend actor duplicates the query, sends it to the two replicated server actors and creates a fresh actor to handle their response. Each of them handles its message and answers to the fresh actor.

This fresh actor considers the first answer and deletes the second. Finally the receptor actor receives its message and dies.

B.2 ANALYSIS

The system is analyzed on a 2GHz Intel Core 2 Duo with 1 Go of RAM. It takes 52 seconds to compute the least fixed point in 32 iterations.

We give here the detail of computed properties. In these properties, we can observe no remaining messages will be pending in the receptor actor mailbox or in the fresh actor mailbox.

We present the abstract element as given by the analyzer. The only modification was to render the abstract elements more readable.

B.2.1 *Control flow abstraction*

The following table describes the control flow properties obtained. Each program point is associated to the properties satisfied by all threads on this program point.

Example B.1 The replicating server.

```

vreceipt1, vserv2, vdupserv3, vclient4,
  serv ▷5 [m6(c) = ζ(e, s)(e ▷7 s || c ◁8 reply());
    reify9(c, ack) = ζ(e, s)(c ◁10 state_serv(e, s, ack))]
||dupserv ▷11 [state_serv12(ego, self, ack) = ζ(e, s)(
  va13, vb14, a ▷15 self || b ▷16 self
||ego ▷17 [m18(c) = ζ(e, s)(
  vf19, a ◁20 m(f) || b ◁21 m(f)
  ||f ▷22 [reply23() = ζ(e, s)(
    c ◁24 reply() || e ▷25 [reply26() = ζ(e, s)(0)]]]
  ||e ▷27 s)]
  ||ack ◁28 available()
  ||e ▷29 s)]
||client ▷30 [init31() = ζ(e, s)(
  serv ◁32 reify(dupserv, e)
  ||e ▷33 [available34() = ζ(e, s')(serv ◁35 m(recept))]]]
||recept ▷36 [reply37() = ζ(e, s)(0)]
||client ◁38 init()

```

In that example, the graphs of equality and inequality among values or marker do not give supplementary information and are not represented. The last column gives a regular approximation of values.

Pp	Interface	Shape
5	{serv}	I : ϵ serv : (2, ϵ)
6	\emptyset	I : ϵ
7	{e, s}	I : 15 + 16 e : (13, ϵ) + (14, ϵ) s : (5, ϵ)
8	{c}	I : 15 + 16 c : (19, ϵ)
9	\emptyset	I : ϵ
10	{e, s, ack, c}	I : ϵ e : (2, ϵ) s : (5, ϵ) ack : (4, ϵ) c : (3; ϵ)
11	{dupserv}	I : ϵ dupserv : (3, ϵ)
12	\emptyset	I : ϵ
15	{self, a}	I : ϵ self : (5, ϵ) a : (13, ϵ)
16	{self, b}	I : ϵ self : (5, ϵ) b : (14, ϵ)
17	{ego, a, b}	I : ϵ ego : (2, ϵ) a : (13, ϵ) b : (14, ϵ)
18	{a, b}	I : ϵ a : (13, ϵ) b : (14, ϵ)

Pp	Interface	Shape
20	{a, f}	I : ϵ a : (13, ϵ) f : (19, ϵ)
21	{b, f}	I : ϵ b : (14, ϵ) f : (19, ϵ)
22	{f, c}	I : ϵ c : (1, ϵ) f : (19, ϵ)
23	{c}	I : ϵ c : (1, ϵ)
24	{c}	I : ϵ c : (1, ϵ)
25	{e}	I : ϵ e : (19, ϵ)
26	\emptyset	I : ϵ
27	{e, s}	I : ϵ e : (2, ϵ) s : (17, ϵ)
28	{ack}	I : ϵ ack : (4, ϵ)
29	{e, s}	I : ϵ e : (3, ϵ) s : (11, ϵ)
30	{recept, serv, dupserv, client}	I : ϵ recept : (1, ϵ) serv : (2, ϵ) dupserv : (3, ϵ) client : (4, ϵ)
31	{recept, serv, dupserv}	I : ϵ recept : (1, ϵ) serv : (2, ϵ) dupserv : (3, ϵ)

Pp	Interface	Shape
32	{serv, dupserv, e}	I : ϵ serv : (2, ϵ) dupserv : (3, ϵ) e : (4, ϵ)
33	{recept, serv, e}	I : ϵ recept : (1, ϵ) serv : (2, ϵ) e : (4, ϵ)
34	{recept, serv}	I : ϵ recept : (1, ϵ) serv : (2, ϵ)
35	{recept, serv}	I : ϵ recept : (1, ϵ) serv : (2, ϵ)
36	{recept}	I : ϵ recept : (1, ϵ)
37	{recept} \emptyset	I : ϵ recept : (1, ϵ)
38	{client}	I : ϵ client : (4, ϵ)

B.2.2 Global occurrence counting abstraction

The global numerical properties are the following.

$H38 : \llbracket 0, 1 \rrbracket$ $H37 : \llbracket 1, 1 \rrbracket$ $H36 : \llbracket 0, 1 \rrbracket$ $H35 : \llbracket 0, 1 \rrbracket$ $H34 : \llbracket 0, 1 \rrbracket$ $H33 : \llbracket 0, 1 \rrbracket$ $H32 : \llbracket 0, 1 \rrbracket$
 $H31 : \llbracket 1, 1 \rrbracket$ $H30 : \llbracket 0, 1 \rrbracket$ $H29 : \llbracket 0, 1 \rrbracket$ $H28 : \llbracket 0, 1 \rrbracket$ $H27 : \llbracket 0, 1 \rrbracket$ $H26 : \llbracket 0, 1 \rrbracket$ $H25 : \llbracket 0, 1 \rrbracket$
 $H24 : \llbracket 0, 1 \rrbracket$ $H23 : \llbracket 0, 1 \rrbracket$ $H22 : \llbracket 0, 1 \rrbracket$ $H21 : \llbracket 0, 2 \rrbracket$ $H20 : \llbracket 0, 2 \rrbracket$ $H18 : \llbracket 0, 1 \rrbracket$ $H17 : \llbracket 0, 1 \rrbracket$
 $H16 : \llbracket 0, 2 \rrbracket$ $H15 : \llbracket 0, 2 \rrbracket$ $H12 : \llbracket 1, 1 \rrbracket$ $H11 : \llbracket 0, 1 \rrbracket$ $H10 : \llbracket 0, 1 \rrbracket$ $H9 : \llbracket 1, 1 \rrbracket$ $H8 : \llbracket 0, 2 \rrbracket$
 $H7 : \llbracket 0, 2 \rrbracket$ $H6 : \llbracket 1, 1 \rrbracket$ $H5 : \llbracket 0, 1 \rrbracket$ $R5 : \llbracket 0, 1 \rrbracket$ $R6 : \llbracket 0, 2 \rrbracket$ $R11 : \llbracket 0, 1 \rrbracket$ $R17 : \llbracket 0, 1 \rrbracket$
 $R22 : \llbracket 0, 1 \rrbracket$ $R25 : \llbracket 0, 1 \rrbracket$ $R30 : \llbracket 0, 1 \rrbracket$ $R33 : \llbracket 0, 1 \rrbracket$ $R36 : \llbracket 0, 1 \rrbracket$

$$H38 + R30 = 1$$

$$H36 + R36 = 1$$

$$H34 - R30 = 0$$

$$H37 = 1$$

$$H35 + R17 - R33 = 0$$

$$H33 - R30 + R33 = 0$$

$$\begin{array}{l}
H32 + R5 - R30 = 0 \\
H30 + R30 = 1 \\
H28 - R11 + R33 = 0 \\
H26 - R22 = 0 \\
H24 - R22 + R36 = 0 \\
H22 - R17 + R22 = 0 \\
H20 - H15 + R11 - R17 = 0 \\
H17 - R11 + R17 = 0 \\
H12 = 1 \\
H10 - R5 + R11 = 0 \\
H8 - R6 + R22 + R25 = 0 \\
H6 = 1 \\
R7, R8, R9, R10, R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, \\
R32, R34, R35, R37, R38 = 0
\end{array}
\quad
\begin{array}{l}
H31 = 1 \\
H29 - R11 = 0 \\
H27 - R17 = 0 \\
H25 - R22 + R25 = 0 \\
H23 - R17 = 0 \\
H21 + H15 + R6 - R11 - R17 = 0 \\
H18 - R11 = 0 \\
H16 + H15 + R6 - 2 * R11 = 0 \\
H11 + R11 = 1 \\
H9 = 1 \\
H7 - R6 = 0 \\
H5 + R5 = 1
\end{array}$$

B.2.3 Linearity abstraction

Binder modes

$$(1, \bullet), (2, \bullet), (3, \bullet), (4, \bullet), (13, \bullet), (14, \bullet), (19, \bullet)$$

Abstract environments

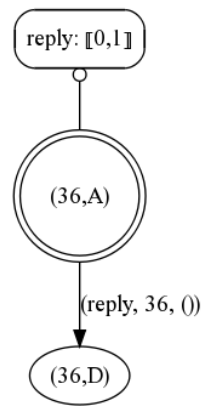
$$\begin{array}{ll}
5 : (\text{serv}, \bullet) & 7 : (e, \bullet), (s, \circ) \\
8 : (c, \circ) & 10 : (e, \bullet), (s, \circ), (\text{ack}, \circ), (c, \circ) \\
11 : (\text{dupserv}, \bullet) & 15 : (\text{self}, \circ), (a, \bullet) \\
16 : (\text{self}, \circ), (b, \bullet) & 17 : (\text{ego}, \bullet), (a, \circ), (b, \circ) \\
18 : (a, \circ), (b, \circ) & 20 : (a, \circ), (f, \circ) \\
21 : (b, \circ), (f, \circ) & 22 : (c, \circ), (f, \bullet) \\
23 : (c, \circ) & 24 : (c, \circ) \\
25 : (e, \bullet) & 27 : (e, \bullet), (s, \circ) \\
28 : (\text{ack}, \circ) & 29 : (e, \bullet), (s, \circ) \\
36 : (\text{recept}, \bullet) & 31 : (\text{recept}, \circ), (\text{serv}, \circ), (\text{dupserv}, \circ) \\
32 : (\text{serv}, \circ), (\text{dupserv}, \circ), (e, \circ) & 33 : (\text{recept}, \circ), (\text{serv}, \circ), (e, \bullet) \\
34 : (\text{recept}, \circ), (\text{serv}, \circ) & 35 : (\text{recept}, \circ), (\text{serv}, \circ) \\
30 : (\text{recept}, \circ), (\text{serv}, \circ), (\text{dupserv}, \circ), (\text{client}, \bullet) & 38 : (\text{client}, \circ)
\end{array}$$

B.2.4 Partitioned abstraction

The next table presents the results given by the partitioned abstract domain. Each binder is associated to an occurrence abstraction and to the interface and mailboxes approximation.

Binder 1 ν recept

Figure B.1 Interface and mailbox abstraction for the binder 1.



MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.1.

OCCURRENCE The local occurrence counting element describes only occurrences of threads specific to the binder 1.

$$\begin{array}{lllll} \text{H36} : [0, 1] & \text{H24} : [0, 1] & \text{R5} : [0, 1] & \text{R6} : [0, 2] & \text{R11} : [0, 1] \\ \text{R17} : [0, 1] & \text{R22} : [0, 1] & \text{R25} : [0, 1] & \text{R30} : [0, 1] & \text{R33} : [0, 1] \\ \text{R36} : [0, 1] & & & & \end{array}$$

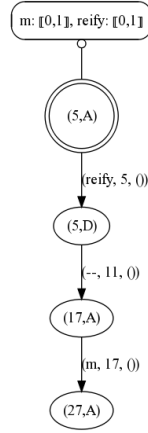
$$\text{H36} + \text{R36} = 1$$

$$\text{H24} - \text{R22} + \text{R36} = 0$$

$$\begin{array}{l} \text{H38, H37, H35, H34, H33, H32, H31, H30, H29, H28, H27, H26, H25, H23, H22, H21,} \\ \text{H20, H18, H17, H16, H15, H12, H11, H10, H9, H8, H7, H6, H5, R7, R8, R9,} \\ \text{R10, R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34,} \\ \text{R35, R37, R38} = 0 \end{array}$$

Binder 2 vserv

Figure B.2 Interface and mailbox abstraction for the binder 2.



MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.2. In that case, as explained in the Chapters 7 and 8, we fail at proving that there is no orphan messages. The production of the message labeled m outside of its interacting unit – on address $(2, \epsilon)$ – without the launching of an actor, gives us this over-approximation. The checking algorithm will not be able to prove the absence of orphans since a message m can be (is) present at the dead node $(5, D)$.

To handle such cases, we will need to rely on other unit approximation in order to guarantee the actor $(17, A)$ to be produced. This look like feasible with the already computed properies but has not been investigated yet.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 2.

$$\begin{array}{lllll}
 H35 : [0, 1] & H32 : [0, 1] & H27 : [0, 1] & H17 : [0, 1] & H5 : [0, 1] \\
 R5 : [0, 1] & R6 : [0, 2] & R11 : [0, 1] & R17 : [0, 1] & R22 : [0, 1] \\
 R25 : [0, 1] & R30 : [0, 1] & R33 : [0, 1] & R36 : [0, 1] &
 \end{array}$$

$$H35 + R17 - R33 = 0$$

$$H32 + R5 - R30 = 0$$

$$H27 - R17 = 0$$

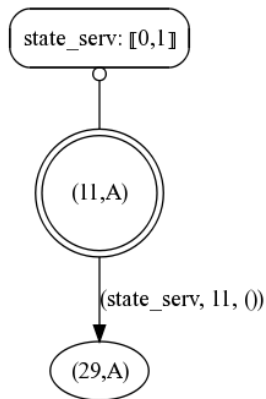
$$H17 - R11 + R17 = 0$$

$$H5 + R5 = 1$$

H38, H37, H36, H34, H33, H31, H30, H29, H28, H26, H25, H24, H23, H22, H21, H20,
 H18, H16, H15, H12, H11, H10, H9, H8, H7, H6, R7, R8, R9, R10, R12, R15,
 R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35, R37, R38 = 0

Binder 3 vdupserv

Figure B.3 Interface and mailbox abstraction for the binder 3.



MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.3.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 3.

H29 : [[0, 1]] H11 : [[0, 1]] H10 : [[0, 1]] R5 : [[0, 1]] R6 : [[0, 2]]
 R11 : [[0, 1]] R17 : [[0, 1]] R22 : [[0, 1]] R25 : [[0, 1]] R30 : [[0, 1]]
 R33 : [[0, 1]] R36 : [[0, 1]]

$$H29 - R11 = 0$$

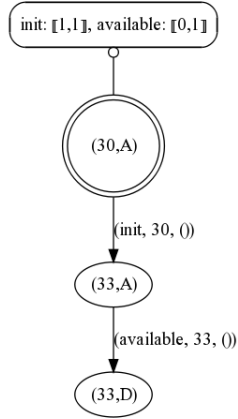
$$H11 + R11 = 1$$

$$H10 - R5 + R11 = 0$$

H38, H37, H36, H35, H34, H33, H32, H31, H30, H28, H27, H26, H25, H24, H23, H22,
 H21, H20, H18, H17, H16, H15, H12, H9, H8, H7, H6, H5, R7, R8, R9, R10,
 R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35,
 R37, R38 = 0

Binder 4 vclient

Figure B.4 Interface and mailbox abstraction for the binder 4.



MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.4.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 4.

$$\begin{array}{lllll}
 \text{H38} : [0, 1] & \text{H33} : [0, 1] & \text{H30} : [0, 1] & \text{H28} : [0, 1] & \text{R5} : [0, 1] \\
 \text{R6} : [0, 2] & \text{R11} : [0, 1] & \text{R17} : [0, 1] & \text{R22} : [0, 1] & \text{R25} : [0, 1] \\
 \text{R30} : [0, 1] & \text{R33} : [0, 1] & \text{R36} : [0, 1] & &
 \end{array}$$

$$\text{H38} + \text{R30} = 1$$

$$\text{H33} - \text{R30} + \text{R33} = 0$$

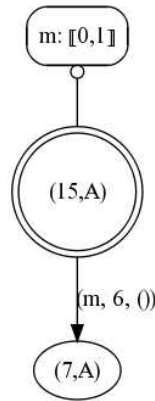
$$\text{H30} + \text{R30} = 1$$

$$\text{H28} - \text{R11} + \text{R33} = 0$$

$$\begin{array}{l}
 \text{H37, H36, H35, H34, H32, H31, H29, H27, H26, H25, H24, H23, H22, H21, H20, H18,} \\
 \text{H17, H16, H15, H12, H11, H10, H9, H8, H7, H6, H5, R7, R8, R9, R10, R12,} \\
 \text{R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35, R37,} \\
 \text{R38} = 0
 \end{array}$$

Binder 13 νa

Figure B.5 Interface and mailbox abstraction for the binder 13.



MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.5.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 13.

$$\begin{array}{lllll}
 H20 : [0, 1] & H15 : [0, 1] & H7 : [0, 1] & R5 : [0, 1] & R6 : [0, 2] \\
 R11 : [0, 1] & R17 : [0, 1] & R22 : [0, 1] & R25 : [0, 1] & R30 : [0, 1] \\
 R33 : [0, 1] & R36 : [0, 1] & & &
 \end{array}$$

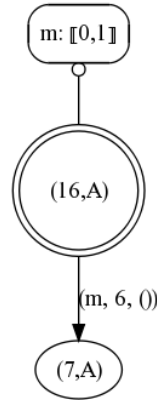
$$H20 + H7 - R17 = 0$$

$$\begin{array}{l}
 H38, H37, H36, H35, H34, H33, H32, H31, H30, H29, H28, H27, H26, H25, H24, H23, \\
 H22, H21, H18, H17, H16, H12, H11, H10, H9, H8, H6, H5, R7, R8, R9, R10, \\
 R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35, \\
 R37, R38 = 0
 \end{array}$$

Binder 14 νb

MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.6.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 14.

Figure B.6 Interface and mailbox abstraction for the binder 14.

$$\begin{array}{lllll}
 H21 : [0, 1] & H16 : [0, 1] & H7 : [0, 1] & R5 : [0, 1] & R6 : [0, 2] \\
 R11 : [0, 1] & R17 : [0, 1] & R22 : [0, 1] & R25 : [0, 1] & R30 : [0, 1] \\
 R33 : [0, 1] & R36 : [0, 1] & & &
 \end{array}$$

$$H21 + H7 - R17 = 0$$

$$\begin{array}{l}
 H38, H37, H36, H35, H34, H33, H32, H31, H30, H29, H28, H27, H26, H25, H24, H23, \\
 H22, H20, H18, H17, H15, H12, H11, H10, H9, H8, H6, H5, R7, R8, R9, R10, \\
 R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35, \\
 R37, R38 = 0
 \end{array}$$

Binder 19 νf

MAILBOX The abstract element obtained for the interface and mailbox abstraction is given in Figure B.7.

OCCURRENCE This local occurrence counting element describes only occurrences of threads specific to the binder 19.

$$\begin{array}{lllll}
 H25 : [0, 1] & H22 : [0, 1] & H8 : [0, 2] & R5 : [0, 1] & R6 : [0, 2] \\
 R11 : [0, 1] & R17 : [0, 1] & R22 : [0, 1] & R25 : [0, 1] & R30 : [0, 1] \\
 R33 : [0, 1] & R36 : [0, 1] & & &
 \end{array}$$

$$H25 - R22 + R25 = 0$$

$$H8 - R6 + R22 + R25 = 0$$

$$\begin{array}{l}
 H38, H37, H36, H35, H34, H33, H32, H31, H30, H29, H28, H27, H26, H24, H23, H21, \\
 H20, H18, H17, H16, H15, H12, H11, H10, H9, H7, H6, H5, R7, R8, R9, R10, \\
 R12, R15, R16, R18, R20, R21, R23, R24, R26, R27, R28, R29, R31, R32, R34, R35, \\
 R37, R38 = 0
 \end{array}$$

Figure B.7 Interface and mailbox abstraction for the binder 19.

