

THÈSE

présentée à

L'ÉCOLE POLYTECHNIQUE

pour obtenir le titre de

DOCTEUR DE L'ÉCOLE POLYTECHNIQUE

spécialité :

INFORMATIQUE

par

Cédric FOURNET

Sujet de la thèse :

**LE JOIN-CALCUL :
UN CALCUL POUR LA PROGRAMMATION
RÉPARTIE ET MOBILE**

*The Join-Calculus:
a Calculus for Distributed Mobile Programming*

Soutenue le 23 Novembre 1998 devant le jury composé de :

MM.	Robin	Milner	<i>Président</i>
	Roberto	Amadio	<i>Rapporteurs</i>
	Gérard	Boudol	
	Jean-Jacques	Lévy	<i>Directeur de thèse</i>
	Gérard	Berry	<i>Examineurs</i>
	Luca	Cardelli	
	Georges	Gonthier	

Remerciements

- M. Robin Milner a bien voulu présider le jury ; je l'en remercie chaleureusement.
- MM. Roberto Amadio et Gérard Boudol ont lu cette thèse en détail et ont accepté d'en être les rapporteurs. Qu'ils en soient remerciés, et qu'ils me pardonnent ses longueurs.
- M. Jean-Jacques Lévy a été un directeur de thèse amical, disponible, et de bon conseil. Il m'a persuadé de l'intérêt de la recherche en informatique et m'a suggéré l'étude de la programmation répartie. Au cours de la thèse, il m'a apporté son soutien tout en me laissant une grande liberté. Je lui en suis particulièrement reconnaissant.
- M. Georges Gonthier a également guidé cette thèse. Sa collaboration fut essentielle à la plupart des résultats décrits ici. Je l'en remercie tout particulièrement.
- MM. Luca Cardelli et Gérard Berry ont accepté de participer au jury. Je les remercie de l'attention qu'ils accordent à mon travail.

Je tiens à exprimer ma gratitude envers Martin Abadi, Michele Boreale, Georges Gonthier, Cosimo Laneve, Jean-Jacques Levy, Luc Maranget, et Didier Rémy, avec qui j'ai eu la chance de collaborer dans l'étude du join-calcul. Je remercie encore Luc Maranget, qui a réalisé avec moi une implémentation répartie du join-calcul. Notre prototype n'aurait sans doute pas abouti sans son bon sens ni son expérience de la compilation. Peter Sewell et Carolina Lavatelli ont relu le manuscrit. Je les remercie de leur patience.

J'ai bénéficié de vives discussions sur les langages de programmation, les calculs de processus, et leur application à la programmation parallèle ou répartie. Merci à Ilaria Castellani, Damien Doligez, Andrew Gordon, Florent Guillaume, Matthew Hennessy, Kohei Honda, Ole Jensen, Fabrice Le Fessant, Xavier Leroy, Ugo Montanari, Uwe Nestmann, Benjamin Pierce, Jon Riecke, Davide Sangiorgi, Peter Sewell, David Turner, et à ceux que j'oublie ici.

Cette thèse s'est déroulée dans l'excellent environnement scientifique et la bonne ambiance du projet PARA et des projets voisins, à l'Institut National de Recherche en Informatique et en Automatique (INRIA). J'ai été invité par MM. Robin Milner et Benjamin Pierce à l'université de Cambridge pendant l'été 1995. J'ai aussi bénéficié des programmes de recherche européens ESPRIT Basic Research Action 6454 - CONFER, puis ESPRIT Working Group 21836 - CONFER-2.

Ce document, ainsi que l'implémentation répartie du join-calcul, ont été produits à l'aide des logiciels Emacs, T_EX , L^AT_EX , et Objective Caml.

Contents

Résumé en français	7
Main Notations	37
1 Introduction	39
1.1 Structure of the dissertation	41
1.2 Related work on the join-calculus	42
2 The Join-Calculus	45
2.1 The channel abstraction	46
2.2 Chemical machines	50
2.2.1 The chemical metaphor	50
2.2.2 The chemical abstract machine	52
2.2.3 Ensuring locality, adding reflexion	54
2.3 The reflexive chemical abstract machine	56
2.3.1 Overview	56
2.3.2 Syntax and scopes	57
2.3.3 Operational semantics	60
2.4 Examples	61
2.4.1 Some wiring	61
2.4.2 Chemical inertness, units, and deadlocks	62
2.4.3 Abstractions of processes	63
2.4.4 Channels of the π -calculus	64
2.4.5 Representing choice	64
2.4.6 The reference cell	65
2.5 Basic properties of the reflexive CHAM	65
2.5.1 Normal forms	66
2.5.2 Built-in locality	66
2.5.3 Reductions on processes	67
2.5.4 Case analyses on reduction steps	67
2.5.5 SOS-style semantics	67
2.5.6 Refined chemical machines	70
2.6 Other models of concurrency	70
2.6.1 Higher-order Gamma	71
2.6.2 Data flow languages	71
2.6.3 Multi-functions	71
2.6.4 Petri nets	72

2.6.5	Other variants of the π -calculus	72
3	Adding Types and Functions	73
3.1	Polymorphism in the join-calculus	74
3.2	The typed join-calculus	75
3.2.1	Syntax	75
3.2.2	Typing rules	76
3.2.3	External primitives	76
3.2.4	Types and chemistry	78
3.2.5	Types at work	80
3.3	Correctness of the evaluation	82
3.3.1	Basic properties for the typing	82
3.3.2	Subject reduction	82
3.3.3	No run-time errors	85
3.4	Functional constructs	85
3.4.1	Sequential control in the join-calculus	86
3.4.2	Two evaluation strategies of the λ -calculus	87
3.4.3	Synchronous names	89
3.4.4	A typed CPS encoding	90
3.4.5	Toward a concurrent functional language	91
3.4.6	Types and side effects	92
3.5	Concurrent objects as join-definitions	93
3.5.1	Primitive objects	94
3.5.2	Values, classes and inheritance	96
3.6	Related type systems for concurrent languages	96
3.6.1	Typing communication patterns	97
3.6.2	Implicit polymorphism and control	97
4	Equivalences and Proof Techniques	99
4.1	Reduction-based semantics	101
4.1.1	Abstract reduction systems	101
4.1.2	What can be observed in the join-calculus	102
4.1.3	Contexts and congruence properties	103
4.1.4	Weak semantics	104
4.1.5	On barbs and contexts	105
4.2	Testing semantics	106
4.3	Fair testing	107
4.3.1	Fair testing versus may testing	109
4.4	Barbed Congruence	110
4.4.1	Diagrams	112
4.4.2	About co-inductive definitions	112
4.4.3	The two congruences	113
4.4.4	Single-barbed bisimulation	115
4.5	Coupled simulations	115
4.5.1	Internal choice and gradual Commitment	116
4.5.2	The two congruences yield distinct equivalences	118
4.5.3	A model of coupled simulations	120

4.6	A hierarchy of equivalences	122
4.6.1	The situation in the π -calculus	122
4.7	Techniques of bisimulation “up to”	125
4.7.1	Confluence by decreasing diagrams	126
4.7.2	Weak bisimulation up to bisimilarity	127
4.7.3	Expansions	128
4.7.4	Accommodating deterministic reductions	131
4.8	Barbed equivalence versus barbed congruence	132
4.8.1	Double-barbed bisimulation	132
4.8.2	All integers on two exclusive barbs	134
4.8.3	A join-calculus interpreter for the join-calculus	137
4.8.4	Reducing contexts to integers	140
4.8.5	Universal contexts	141
5	The Open Join-Calculus	145
5.1	Opening the calculus	147
5.1.1	Open syntax	147
5.1.2	Open chemistry	148
5.1.3	Tracking transitions in context	151
5.2	Weak bisimulation	153
5.2.1	Renaming and congruence properties	154
5.3	Asynchronous bisimulation	158
5.3.1	The J-open RCHAM	159
5.3.2	Asynchronous bisimulation	160
5.3.3	Ground bisimulations	162
5.4	Reduction-based equivalences on open terms	163
5.4.1	Observation	163
5.4.2	Extruded names and congruence properties	164
5.4.3	Plug-in’s versus extruded names	165
5.4.4	Weak bisimulation versus barbed congruence	167
5.5	The discriminating power of name comparison	168
5.5.1	Should the join-calculus provide name comparison?	168
5.5.2	The join-calculus with name-matching	168
5.5.3	Barbed congruence is a weak bisimulation	169
5.6	Related equivalences in the π -calculus	173
6	Encodings	177
6.1	On encodings	178
6.1.1	Formal properties of translations	179
6.1.2	Contexts and compositionality	180
6.2	The core join-calculus	181
6.3	Simpler definitions	182
6.3.1	Binders and internal state	182
6.3.2	Rearranging synchronization patterns	183
6.3.3	Encoding complex definitions	185
6.4	Relays everywhere	188
6.5	Polyadic messages	193

6.5.1	Communicating pairs and lists	194
6.5.2	Compositional translation	194
6.5.3	Full abstraction	195
6.6	Cross-encodings with the π -calculus	199
6.6.1	The asynchronous π -calculus	200
6.6.2	Asynchrony, relays, and equators	201
6.6.3	Encoding the π -calculus	202
6.6.4	Encoding the join-calculus	205
6.7	Proof of Theorem 10	209
7	Locality, Migration, and Failures	221
7.1	Computing with locations	222
7.1.1	Distributed solutions	223
7.1.2	Should locations be nested?	226
7.1.3	The location tree	227
7.1.4	Moving locations	229
7.1.5	Examples of agent-based protocols	229
7.1.6	Circular migration	233
7.1.7	Erasing locality information	233
7.2	Failure and recovery	236
7.2.1	The fail-stop model	236
7.2.2	Representing failures	237
7.2.3	Primitives for failure and recovery	240
7.2.4	Fault-tolerant protocols	241
7.2.5	Other models for failure	244
7.3	Proofs for mobile protocols	246
7.3.1	A few simplifying equations	247
7.3.2	Failures and atomicity	250
7.4	Related work	251
7.4.1	Applets in Java	251
7.4.2	Migration as a programming language feature	251
7.4.3	Migration as a programming paradigm	252
7.4.4	Agent-based mobility and network transparency	252
7.4.5	Locality and failures	252
7.4.6	Modeling heterogeneous networks	253
	Conclusions	255
	References	257

List of Figures

2.1	Syntax for the join-calculus	59
2.2	Scopes for the join-calculus	59
2.3	The reflexive chemical machine (RCHAM)	60
2.4	Structural congruence on processes and definitions	69
2.5	Syntactic transitions with wide labels	69
3.1	Syntax for the types	77
3.2	Typing rules for the join-calculus	77
3.3	Extended syntax with synchronous names	89
3.4	Syntax for a language with processes and expressions	92
4.1	A hierarchy of equivalences for the join-calculus.	123
5.1	Scopes for the open join-calculus	149
5.2	The open reflexive chemical machine	149
5.3	The J-open RCHAM	159
7.1	Syntax for the distributed-join-calculus	237
7.2	Scopes for the distributed-join-calculus	238
7.3	The distributed reflexive chemical machine	239

Main Notations

In this dissertation, we use the following standard conventions:

Tuples: $\tilde{v}$ is the tuple $v_1, v_2, \dots, v_n$ for some integer $n \geq 0$; when the notation occurs several times in the same formula, we assume a different choice of n for each argument of the notation.

Multisets: We use the same notations for both sets and multisets—the context should prevent any ambiguity. We write $\{\tilde{x}\}$ for the set or the multiset that contains the $\tilde{x}$'s, and $\{x \mid p(x)\}$ to denote the set or the multiset that contains all the x 's such that the predicate $p(x)$ holds. We use the operators $\cap$, $\cup$ and, $\uplus$ to note intersection, union, and disjoint union of sets and multisets, respectively.

Relations and predicates: A binary relation $\mathcal{R}$ between the sets S_1 and S_2 is a subset of $S_1 \times S_2$. Most of our relations will range over the same sets. We usually adopt an infix notation for all relations. We let the variables $\mathcal{R}, \mathcal{R}', \varphi$ and symbols of equality and equivalences range over relations. We write Id for the identity relation.

Let $\mathcal{R}$ and $\mathcal{R}'$ be two relations. When defined, we write $\mathcal{R}\mathcal{R}'$ for the composition of relations $\{(x, y) \mid \exists z, x \mathcal{R} z \mathcal{R}' y\}$, $\mathcal{R}^{-1}$ for the converse relation $\{(y, x) \mid x \mathcal{R} y\}$, $\mathcal{R}^n$ for the repeated relation inductively defined by $\mathcal{R}^0 = Id$ and $\mathcal{R}^{n+1} = \mathcal{R}\mathcal{R}^n$, $\mathcal{R}^=$ for the reflexive closure $Id \cup \mathcal{R}$, $\mathcal{R}^+$ for the transitive closure $\bigcup_{n \geq 1} \mathcal{R}^n$, and $\mathcal{R}^*$ for the reflexive-transitive closure $\bigcup_{n \geq 0} \mathcal{R}^n$.

A preorder is a transitive reflexive relation; an equivalence is a symmetric transitive reflexive relation. A relation refines another relation when it is included in it.

Every relation $\mathcal{R}$ defines a predicate, also written $\mathcal{R}$, defined as $x \mathcal{R} y$ iff $\exists y \mid x \mathcal{R} y$. For every predicate T , the predicate $\neg T$ is its negation.

Strings: A string is a finite sequence of elements from a base set. We let the variables φ, σ range over strings. $\varphi\sigma$ is the concatenation of the strings φ and σ , and ϵ is the empty string.

Variables and Substitutions: We use variables to denote names and provide scoping rules that determine when a variable is bound. Bound variables can be substituted for any variable that does not appear in its scope; we call such internal substitutions α -conversion. A variable is *fresh* with regards to a collection of terms when it does not appear in any of these terms.

We use a postfix notation for substitutions. We write $\{y^1/x_1, \dots, y^n/x_n\}$ or simply $\{\tilde{y}/\tilde{x}\}$ for the substitution that simultaneously replaces every occurrence of a variable x_i by the variable y_i , and write σ for an arbitrary substitution. We assume implicit α -conversion on bound variables before substitution to avoid name clashes.