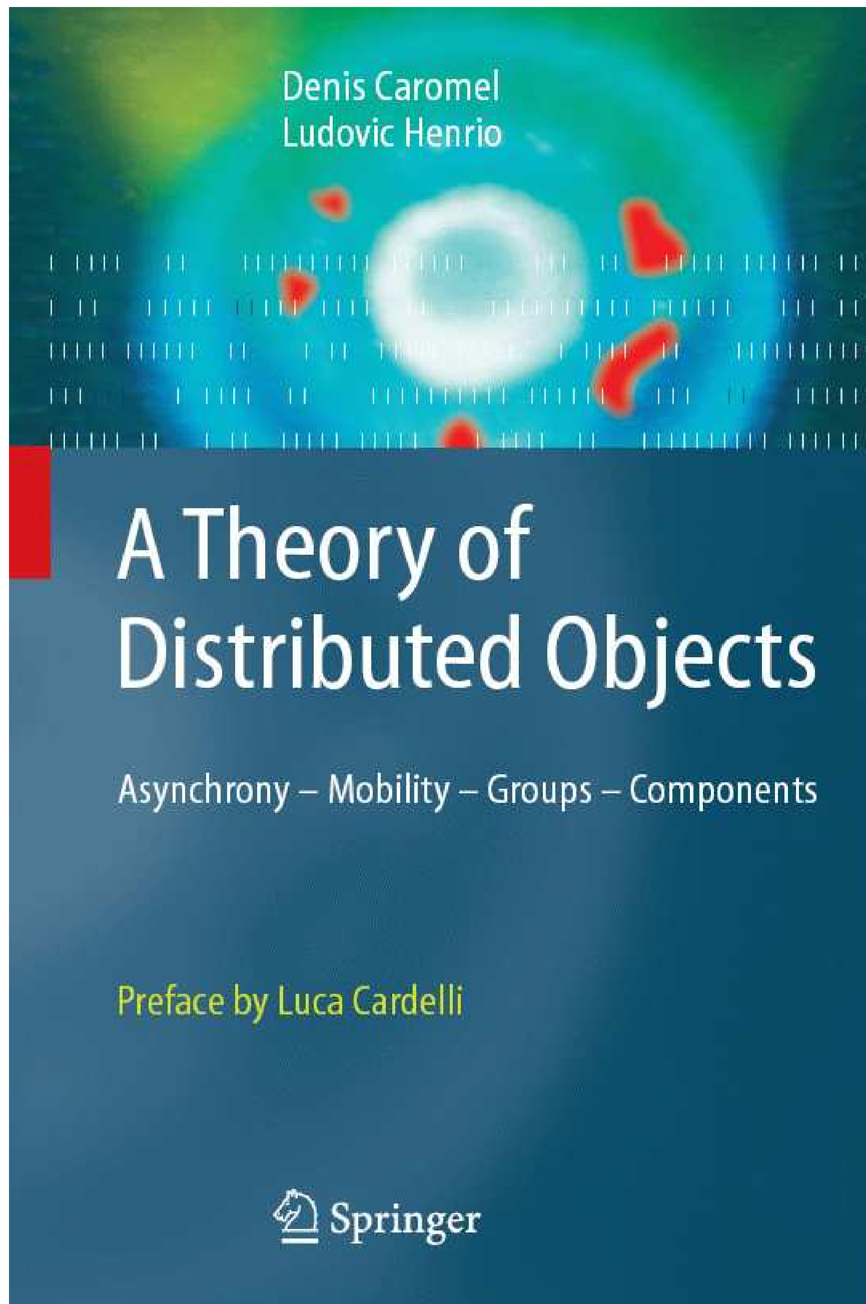
Teaching Material For
A Theory of Distributed Objects

Denis Caromel Ludovic Henrio
Inria Sophia-Antipolis {caromel,henrio}@sophia.inria.fr

ISBN: 3-540-20866-6



To Isa, Ugo, Tom,
Taken from us by the Tsunami, Sri Lanka, December 26, 2004

Isabelle, my wife, my lover, my fellow intellect, I miss you so badly.
My soul, my body, my brain, all hurt for you, all cry out for you.
Your smile, your spirit would bring joy and light to all around you.
Your plans were to do voluntary work to help humanity,
I know you would have given courage and cheer to so many.

Ugo, my 8 year old boy, you could not wait to understand the world.
You even found your own definition of infinity: God!
I will remember forever when you would call me "Papaa ? ..."
with that special tone, to announce a tough question.

Tom, my 5 year old boy, you could fight so hard and yet be so sweet.
You were so strong, and you could be so gentle.
Your determination was impressive, but clearly becoming thoughtful.
I will remember forever when after a fight,
you would jump up on my lap and give me a sweet, loving hug.

So many years of happiness and joy,
May your spirits be with us and in me forever

Denis,
Nice hospital,
January 6, 2005

To Françoise, Marc, Laurianne and Sébastien,
and my precious friends

Ludovic,
December 10, 2004

Preface

With the advent of wide-area networks such as the Internet, distributed computing has to expand from its origins in shared-memory computing and local-area networks to a wider context. A large part of the additional complexity is due to the need to manage asynchrony, which is an unavoidable aspect of high-latency networks. Harnessing asynchronous communications is still an open area of research.

This monograph studies a natural programming model for distributed object-oriented programming. In this model, objects make asynchronous method invocations to other objects, and then concurrently carry on until the results of the requests are needed. Only at that point may they have to wait for the results to be completely computed; this delayed wait is called wait-by-necessity. Aspects of such a model have been proposed and formalized in the past: futures have been built into early concurrent languages, and various distributed object calculi have been investigated. However, this is the first time the two features, futures and distributed objects, have been studied formally together.

The result is a natural and disciplined programming model for asynchronous computing, one worthy of study. For example, it is important to understand under which conditions asynchronous execution produces predictable outcomes, without the usual combinatorial explosion of concurrent execution. Even the simplest sequential program becomes highly concurrent under wait-by-necessity execution, and yet such concurrency does not always imply that multiple outcomes are possible. One of the main technical contributions of the monograph, beyond the formalization of the programming model, is a sufficient condition for deterministic evaluation (confluence) of programs.

This monograph addresses problems that have been long identified as fundamental stumbling blocks in writing correct distributed programs. It constitutes a significant step forward, particularly in the area of formalizing and generalizing some of the best ideas proposed so far, coming up with new techniques, and providing a solid foundation for further study. The techniques studied here also have a very practical potential.

Cambridge, 2004-11-15
Luca Cardelli

Caromel · Henrio

A Theory of Distributed Objects

Distributed and communicating objects are becoming ubiquitous. In global, Grid and peer-to-peer computing environments, extensive use is made of objects interacting through method calls. So far, no general formalism has been proposed for the foundation of such systems.

Caromel and Henrio are the first to define a calculus for distributed objects interacting using asynchronous method calls with generalized futures, i.e., wait-by-necessity – a must in large-scale systems, providing both high structuring and low coupling, and thus scalability. The authors provide very generic results on expressiveness and determinism, and the potential of their approach is further demonstrated by its capacity to cope with advanced issues such as mobility, groups, and components.

Researchers and graduate students will find here an extensive review of concurrent languages and calculi, with comprehensive figures and summaries.

Developers of distributed systems can adopt the many implementation strategies that are presented and analyzed in detail.

ISBN 3-540-20866-6

Contents

Preface by Luca Cardelli . . . . **VII**

Table of Contents . . . . **IX**

Lists of Figures, Tables, Definitions and Properties . . . . **XV**

Prologue . . . . **XXV**

Reading Path and Teaching . . . . **XXIX**

Part I Review

1 Analysis **3**
1.1 A Few Definitions 3
1.2 Distribution, Parallelism, Concurrency 3
1.2.1 Parallel Activities 3
1.2.2 Sharing 3
1.2.3 Communication 4
1.2.4 Synchronization 4
1.2.5 Reactive vs. Proactive vs. Synchronous 4
1.3 Objects 5
1.3.1 Object vs. Remote Reference and Communication . . 5
1.3.2 Object vs. Parallel Activity 5
1.3.3 Object vs. Synchronization 5
1.4 Summary and Orientation 5

2 Formalisms and Distributed Calculi **9**
2.1 Basic Formalisms 9
2.1.1 Functional Programming and Parallel Evaluation . . . 10
2.1.2 Actors 10

2.1.3 π-calculus 11
2.1.4 Process Networks 12
2.1.5 ς-calculus 12
2.2 Concurrent Calculi and Languages 13
2.2.1 MultiLisp 13
2.2.2 PICT 13
2.2.3 Ambient Calculus 13
2.2.4 Join-calculus 15
2.2.5 Other Expressions of Concurrency 15
2.3 Concurrent Object Calculi and Languages 15
2.3.1 ABCL 15
2.3.2 Obliq and Øjeblik 22
2.3.3 The $\pi o\beta\lambda$ Language 22
2.3.4 Gordon and Hankin Concurrent Calculus: **conc**ς-calculus 22
2.4 Synthesis and Classification 22

Part II ASP Calculus

3 An Imperative Sequential Calculus 31
3.1 Syntax 31
3.2 Semantic Structures 32
3.2.1 Substitution 32
3.2.2 Store 32
3.2.3 Configuration 32
3.3 Reduction 32
3.4 Properties 32

4 Asynchronous Sequential Processes 35
4.1 Principles 35
4.2 New Syntax 35
4.3 Informal Semantics 36
4.3.1 Activities 36
4.3.2 Requests 36
4.3.3 Futures 36
4.3.4 Serving Requests 36

5 A Few Examples 39
5.1 Binary Tree 39
5.2 Distributed Sieve of Eratosthenes 39
5.3 From Process Networks to ASP 39
5.4 Example: Fibonacci Numbers 39
5.5 A Bank Account Server 39

Part III Semantics and Properties

6 Parallel Semantics **47**
6.1 Structure of Parallel Activities 47
6.2 Parallel Reduction 48
6.2.1 More Operations on Store 48
6.2.2 Reduction Rules 49
6.3 Well-formedness 51

7 Basic ASP Properties **55**
7.1 Notation and Hypothesis 56
7.2 Object Sharing 56
7.3 Isolation of Futures and Parameters 56

8 Confluence Property **59**
8.1 Configuration Compatibility 59
8.2 Equivalence Modulo Future Updates 61
8.2.1 Principles 61
8.2.2 Alias Condition 61
8.2.3 Sufficient Conditions 62
8.3 Properties of Equivalence Modulo Future Updates 63
8.4 Confluence 64

9 Determinacy **67**
9.1 Deterministic Object Networks 67
9.2 Toward a Static Approximation of DON Terms 67
9.3 Tree Topology Determinism 69
9.4 Deterministic Examples 69
9.4.1 The Binary Tree 69
9.4.2 The Fibonacci Number Example 69
9.5 Discussion: Comparing Request Service Strategies 69

Part IV A Few More Features

10 More Confluent Features **73**
10.1 Delegation 73
10.2 Explicit Wait 75
10.3 Method Update 75

11 Non-Confluent Features **77**
11.1 Testing Future Reception 77
11.2 Non-blocking Services 77
11.3 Testing Request Reception 78
11.4 Join Patterns 78
11.4.1 Translating Join Calculus Programs 78

11.4.2 Extended Join Services in ASP 79

12 Migration **81**
12.1 Migrating Active Objects 81
12.2 Optimizing Future Updates 81
12.3 Migration and Confluence 81

13 Groups **83**
13.1 Groups in an Object Calculus 83
13.2 Groups of Active Objects 84
13.3 Groups, Determinism, and Atomicity 84

14 Components **89**
14.1 From Objects to Components 89
14.2 Hierarchical Components 89
14.3 Semantics 90
14.4 Deterministic Components 91
14.5 Components and Groups: Parallel Components 93
14.6 Components and Futures 94

15 Channels and Reconfigurations **95**
15.1 Genuine ASP Channels 95
15.2 Process Network Channels in ASP 95
15.3 Internal Reconfiguration 96
15.4 Event-Based Reconfiguration 96

Part V Implementation Strategies

16 A Java API for ASP: ProActive **99**
16.1 Design and API 99
16.1.1 Basic API and ASP Equivalence 99
16.1.2 Mapping Active Objects to JVMs: Nodes 99
16.1.3 Basic Patterns for Using Active Objects 99
16.1.4 Migration 100
16.1.5 Group Communications 100
16.2 Examples 100
16.2.1 Parallel Binary Tree 100
16.2.2 Eratosthenes 100
16.2.3 Fibonacci 110

17 Future Update **117**
17.1 Future Forwarding 117
17.2 Update Strategies 117
17.2.1 ASP and Generalization: Encompassing All Strategies 117
17.2.2 No Partial Replies and Requests 117

17.2.3 Forward-Based . . . 118
17.2.4 Message-Based . . . 119
17.2.5 Lazy Future Update . . . 120
17.3 Synthesis and Comparison of the Strategies . . . 121

18 Loosing Rendezvous . . . 123
18.1 Objectives and Principles . . . 123
18.2 Asynchronous Without Guarantee . . . 123
18.3 Asynchronous Point-to-Point FIFO Ordering . . . 124
18.4 Asynchronous One-to-All FIFO Ordering . . . 126
18.5 Conclusion . . . 127

19 Controlling Pipelining . . . 129
19.1 Unrestricted Parallelism . . . 129
19.2 Pure Demand Driven . . . 129
19.3 Controlled Pipelining . . . 129

20 Garbage Collection . . . 131
20.1 Local Garbage Collection . . . 131
20.2 Futures . . . 131
20.3 Active Objects . . . 131

Part VI Final Words

21 ASP Versus Other Concurrent Calculi . . . 135
21.1 Basic Formalisms . . . 135
21.1.1 Actors . . . 135
21.1.2 π-calculus and Related Calculi . . . 135
21.1.3 Process Networks . . . 135
21.1.4 ς-calculus . . . 135
21.2 Concurrent Calculi and Languages . . . 135
21.2.1 MultiLisp . . . 135
21.2.2 Ambient Calculus . . . 135
21.2.3 join-calculus . . . 135
21.3 Concurrent Object Calculi and Languages . . . 135
21.3.1 Obliq and Øjeblik . . . 135
21.3.2 The $\pi o\beta\lambda$ Language . . . 135

22 Conclusion . . . 137
22.1 Summary . . . 137
22.2 A Dynamic Property for Determinism . . . 137
22.3 ASP in Practice . . . 137
22.4 Stateful Active Objects vs. Immutable Futures . . . 137
22.5 Perspectives . . . 137

23 Epilogue . . . 139

Appendices

A Equivalence Modulo Future Updates **145**
A.1 Renaming 145
A.2 Reordering Requests ($R_1 \equiv_R R_2$) 145
A.3 Future Updates 145
A.3.1 Following References and Sub-terms 145
A.3.2 Equivalence Definition 147
A.4 Properties of $\equiv_F$ 149
A.5 Sufficient Conditions for Equivalence 150
A.6 Equivalence Modulo Future Updates and Reduction 150
A.7 Another Formulation 151
A.8 Decidability of $\equiv_F$ 151
A.9 Examples 152

B Confluence Proofs **155**
B.1 Context 155
B.2 Lemmas 155
B.3 Local Confluence 156
B.4 Calculus with service based on activity name: $Serve(\alpha)$ . . . 156
B.5 Extension 156

References **161**

Notation **173**

Syntax of ASP Calculus **179**

Operational Semantics **181**

Overview of Properties **183**

Overview of ASP Extensions **185**

List of Figures

1	Suggested reading paths	XXX
2.1	Classification of calculi (informal)	9
2.2	A binary tree in CAML	10
2.3	A factorial actor [9]	11
2.4	Execution of the sieve of Eratosthenes in Process Networks	13
2.5	Sieve of Eratosthenes in Process Networks [100]	14
2.6	Sieve of Eratosthenes in ς-calculus [3]	16
2.7	Binary tree in ς-calculus [3]	16
2.8	A simple Fibonacci example in PICT [130]	18
2.9	A factorial example in the core PICT language [130]	18
2.10	Locks in ambients [47]	18
2.11	Channels in ambients [47]	19
2.12	A cell in the join-calculus [68]	20
2.13	Bounded buffer in ABCL [161]	21
2.14	The three communication types in ABCL	22
2.15	Prime number sieve in Obliq	23
2.16	Binary tree in (a language inspired by) $\pi o \beta \lambda$ [110]	24
2.17	$\pi o \beta \lambda$ parallel binary tree, equivalent to Fig. 2.16 [110]	25
4.1	Objects and activities topology	35
4.2	Example of a parallel configuration	37
5.1	Example: a binary tree	40
5.2	Topology and communications in the parallel binary tree	40
5.3	Example: sieve of Eratosthenes (pull)	41
5.4	Topology of sieve of Eratosthenes (pull)	41
5.5	Example: sieve of Eratosthenes (push)	41
5.6	Topology of sieve of Eratosthenes (push)	42
5.7	Process Network vs. object network	42
5.8	Fibonacci number processes	42

5.9 Example: Fibonacci numbers 43
5.10 Topology of a bank application 43
5.11 Example: bank account server 43

6.1 Example of a deep copy: $copy(\iota, \sigma_\alpha)$ 49
6.2 NEWACT rule 49
6.3 A simple forwarder 50
6.4 REQUEST rule 51
6.5 SERVE rule 52
6.6 ENDSERVICE rule 52
6.7 REPLY rule 53
6.8 Another example of configuration 53

7.1 An informal property diagram 55
7.2 Absence of sharing 56
7.3 Store partitioning: future value, active store, request parameter 57

8.1 Example of RSL 60
8.2 Example of RSL compatibility 61
8.3 Two equivalent configurations modulo future updates . . 62
8.4 An example illustrating the alias condition 63
8.5 Updates in a cycle of futures 64
8.6 Confluence 65
8.7 Confluence without cycle of futures 65

9.1 A non-DON term 68
9.2 Concurrent replies in the binary tree case 70
9.3 Fibonacci number RSLs 70

10.1 Explicit delegation in ASP 74
10.2 Implicit delegation in ASP 75

11.1 A join-calculus cell in ASP 78

12.1 Chain of method calls and chain of corresponding futures 82

13.1 A group of passive objects 83
13.2 Request sending to a group of active objects 85
13.3 An activated group of objects 85
13.4 A confluent program if communications are atomic . . . 86
13.5 Execution with atomic group communications 87
13.6 An execution without atomic group communications . . 88

14.1 A primitive component 90
14.2 Fibonacci as a composite component 91

14.3 A definition of Fibonacci components 92
14.4 Deployment of a composite component 92
14.5 A parallel component using groups 93
14.6 Components and futures 94

15.1 Requests on separate channels do not interfere 95
15.2 A non-deterministic merge 96
15.3 A channel specified with an active object 96

16.1 A simple mobile agent in ProActive 101
16.2 Method call on group 102
16.3 Dynamic typed group of active objects 103
16.4 Sequential binary tree in Java 104
16.5 Subclassing binary tree for a parallel version 105
16.6 Main binary tree program in ProActive 105
16.7 Execution of the parallel binary tree program of Fig. 16.6 106
16.8 Screenshot of the binary tree at execution 107
16.9 Sequential Eratosthenes in Java 108
16.10 Sequential `Prime` Java class 108
16.11 Parallel Eratosthenes in Java ProActive 109
16.12 Parallel `ActivePrime` class 110
16.13 Graph of active objects in the Fibonacci program 111
16.14 Main Fibonacci program in ProActive 111
16.15 The class `Add` of the Fibonacci program 112
16.16 The class `Cons1` of the Fibonacci program 113
16.17 The class `Cons2` of the Fibonacci program 114
16.18 Graphical visualization of the Fibonacci program using IC2D 115

17.1 A future flow example. 118
17.2 General strategy: any future update can occur 119
17.3 No partial replies and requests 119
17.4 Future updates for the forward-based strategy 120
17.5 Message-based strategy: future received and update messages 120
17.6 Lazy future update: only needed futures are updated . . 121
17.7 Future update strategies 121

18.1 Example: activities synchronized by rendezvous 123
18.2 ASP with rendezvous – message ordering 124
18.3 Asynchronous communications without guarantee 125
18.4 Asynchronous point-to-point FIFO communications . . . 126
18.5 Asynchronous one-to-all FIFO communications 127

19.1 Strategies for controlling parallelism 129

23.1 Potential queues, buffering, pipelining, and strategies in ASP 139
23.2 Classification of strategies for sending requests 140

23.3 Classification of strategies for future update 141

A.1 Simple example of future equivalence 146
A.2 The principle of the alias conditions 148
A.3 Simple example of future equivalence 153
A.4 Example of a "cyclic" proof 153
A.5 Equivalence in the case of a cycle of futures 153
A.6 Example of alias condition 154

B.1 SERVE/REQUEST 157
B.2 ENDSERVICE/REQUEST 158
B.3 The diamond property (Property B.12) proof 159

2 Diagram of properties 183

List of Tables

1.1 Aspects of distribution, parallelism, and concurrency . . . 6
1.2 Aspects of ASP 7

2.1 The syntax of an Actors language [9] 10
2.2 Aspects of Actors 11
2.3 The syntax of π-calculus 11
2.4 π-calculus structural congruence 12
2.5 π-calculus reaction rules 12
2.6 Aspects of π-calculus 13
2.7 Aspects of Process Networks 13
2.8 The syntax of **imp**ς-calculus [3] 13
2.9 Well-formed store 14
2.10 Well-formed stack 15
2.11 Semantics of **imp**ς-calculus (big-step, closure based) . . . 15
2.12 Aspects of MultiLisp 17
2.13 A syntax for Pict [132] 17
2.14 Aspects of Pict 19
2.15 The syntax of Ambient calculus 19
2.16 Aspects of Ambients 19
2.17 The syntax of the join-calculus 20
2.18 Main rules defining evaluation in the Join calculus 20
2.19 Aspects of the Join-Calculus 20
2.20 Aspects of ABCL 23
2.21 The syntax of Øjeblik 23
2.22 Aspects of Obliq and Øjeblik 24
2.23 Aspects of $\pi o\beta\lambda$ 24
2.24 The syntax of **conc**ς-calculus [78] 25
2.25 Aspects of **conc**ς-calculus 26
2.26 Summary of a few calculi and languages 27

3.1 Syntax of ASP sequential calculus 31

3.2 Sequential reduction 33

4.1 Syntax of ASP parallel primitives 36
4.2 Syntax of ASP calculus 36

6.1 Deep copy 48
6.2 Parallel reduction (used or modified values are non-gray) . 50

10.1 Rules for delegation (DELEGATE) 73

13.1 Reduction rules for groups 84
13.2 Atomic reduction rules for groups 85

16.1 Relations between ASP constructors and ProActive API . 99
16.2 Migration primitives in ProActive 100

17.1 Generalized future update 118
17.2 No partial replies and requests protocol 118
17.3 Forward-based protocol 119
17.4 Message-based protocol for future update 120

22.1 Duality active objects (stateful) and futures (immutable) 137

A.1 Reordering requests 145
A.2 Path definition 147
A.3 Equivalence rules 152

1 Sequential reduction 181
2 Deep copy 182
3 Parallel reduction (used or modified values are non-gray) . 182

List of Definitions and Properties

Definition	1.1	Parallelism	3
Definition	1.2	Concurrency	3
Definition	1.3	Distribution	3
Definition	1.4	Asynchronous systems	3
Definition	1.5	Future	4
Definition	1.6	Reactive system	4
Definition	1.7	Synchrony hypothesis	4
Definition	1.8	Wait-by-necessity	5
Definition	3.1	Well-formed sequential configuration	32
Definition	3.2	Equivalence on sequential configurations	32
Property	3.3	Well-formed sequential reduction	32
Property	3.4	Determinism	32
Definition	6.1	Copy and merge	48
Property	6.2	Copy and merge	48
Definition	6.3	Future list	54
Definition	6.4	Well-formedness	54
Property	6.5	Well-formed parallel reduction	54
Definition	7.1	Potential services	56
Property	7.2	Store partitioning	56
Definition	8.1	Request Sender List	59
Definition	8.2	RSL comparison $\trianglelefteq$	59
Definition	8.3	RSL compatibility: $RSL(\alpha) \bowtie RSL(\beta)$	59
Definition	8.4	Configuration compatibility: $P \bowtie Q$	59
Definition	8.5	Cycle of futures	63
Definition	8.6	Parallel reduction modulo future updates	63
Property	8.7	Equivalence modulo future updates and reduction	63
Property	8.8	Equivalence and generalized parallel reduction	64
Definition	8.9	Confluent configurations: $P_1 \curlyvee P_2$	64
Theorem	8.10	Confluence	64
Definition	9.1	DON	67

Property 9.2 DON and compatibility . . . 67
Theorem 9.3 DON determinism . . . 67
Definition 9.4 Static DON . . . 68
Property 9.5 Static approximation . . . 69
Theorem 9.6 SDON determinism . . . 69
Theorem 9.7 Tree determinacy, TDON . . . 69
Definition 10.1 Well-formedness . . . 74
Definition 14.1 Primitive component . . . 89
Definition 14.2 Composite component . . . 89
Definition 14.3 Deterministic Primitive Component (DPC) . . . 91
Definition 14.4 Deterministic Composite Component (DCC) . . . 93
Property 14.5 DCC determinism . . . 93
Definition 17.1 Forwarded futures . . . 117
Property 17.2 Origin of futures . . . 117
Property 17.3 Forwarded futures flow . . . 117
Property 17.4 No forwarded futures . . . 117
Property 17.5 Forward-based future update is eager . . . 118
Property 17.6 Message-based strategy is eager . . . 119
Definition 18.1 Triangle pattern . . . 125
Definition A.1 $a \overset{\alpha}{\mapsto}_L b$ . . . 145
Definition A.2 $a \overset{\alpha *}{\mapsto}_L b$ . . . 146
Lemma A.3 $\overset{\alpha}{\mapsto}_L$ and $\overset{\alpha *}{\mapsto}_L$ . . . 146
Lemma A.4 Uniqueness of path destination . . . 147
Definition A.5 Equivalence modulo future updates: $P \equiv_F Q$ . . . 147
Property A.6 Equivalence relation . . . 148
Definition A.7 Equivalence of sub-terms . . . 148
Lemma A.8 Sub-term equivalence . . . 149
Property A.9 Equivalence and compatibility . . . 149
Lemma A.10 $\equiv_F$ and store update . . . 149
Lemma A.11 $\equiv_F$ and substitution . . . 149
Lemma A.12 A characterization of deep copy . . . 149
Lemma A.13 Copy and merge . . . 149
Lemma A.14 $\equiv_F$ and store merge . . . 150
Property A.15 REPLY and $\equiv_F$ . . . 150
Property A.16 Sufficient condition for equivalence . . . 150
Property A.17 $\equiv_F$ and reduction(1) . . . 150
Definition A.18 Parallel reduction modulo future updates . . . 150
Property A.19 $\equiv_F$ and reduction(2) . . . 150
Corollary A.20 $\equiv_F$ and reduction . . . 150
Definition A.21 Equivalence modulo future updates (2) . . . 151
Property A.22 Equivalence of the two equivalence definitions . . . 151
Property A.23 Decidability . . . 151
Property B.1 Confluence . . . 155
Lemma B.2 $\mathcal{Q}$ and compatiblity . . . 155

Lemma B.3 Independent stores 155
Lemma B.4 Extensibility of local reduction 155
Lemma B.5 *copy* and locations 156
Lemma B.6 Multiple copies 156
Lemma B.7 Copy and store update 156
Corollary B.8 Copy and store update 156
Property B.9 Diamond property 156
Lemma B.10 $\equiv_F$ and $\mathcal{Q}(Q, Q')$ 156
Lemma B.11 REPLY vs. other reduction 156
Property B.12 Diamond property with $\equiv_F$ 159

Prologue

Distributed objects are becoming ubiquitous. *Communicating objects* interact at various levels (application objects, Web and middleware services), and in a wide range of environments (mobile devices, local area networks, Grid, and P2P). These objects send messages, call methods on each other's interfaces, and receive requests and replies.

Why would we employ objects to act as interacting entities? An answer with a religious twist would be that *object orientation* has, so far, won the language crusade. However, a technical answer has more substance: objects are stateful abstractions. Any globally-distributed computation must rely on various levels of state, somehow acting as a cache for improved locality, leading to greater scalability and performance. In a multi-tier application server, for instance, objects representing persistent data (e.g., Entity Beans) act as a cache for data within the *n*-tier database.

Thus, stateful objects interact with each other. Why should they communicate with *method calls* rather than with messages traveling over channels? One answer is that this is exactly what objects are all about: distributed systems should not abandon such a critical feature for software structuring. *Remote method invocation* in industrial platforms, following 15 years of research in academia, has taken off, and appears to be a practical and effective solution. Moreover, method calls are also about safety and verification, a highly desirable feature for distributed, multi-principal, multi-domain applications. Because method calls and the interface imply the emergence of *types*, remote method invocations fall within the scope of type theories and practical verifications – including static analyses, which rely heavily on inter-procedural analysis.

With distribution spanning the world ever more widely, an intrinsic characteristic of communication is high latency, with an unbreakable barrier of 70 milliseconds for a signal to go half-way around the world at the speed of light. Large systems, with potentially thousands of interacting entities, cannot accommodate the high coupling induced by synchronous calls, because

such coupling can lead to a blocked chain of remote method calls spanning a large number of entities. An extreme case that requires non-synchronous invocation is the handling of the disconnected mode in wireless settings. In sum, high latency and low coupling call for asynchronous interactions, as in the case of distributed objects: *asynchronous method calls*. But if we want method calls to retain their full capacity, one-way calls on their own are insufficient. Asynchronous method calls with returns are needed, leading to an emerging abstraction: namely, *futures*, the expected result of a given asynchronous method call. Futures turn out to be a very effective abstraction for large distributed systems, preserving both low coupling and high structuring.

To summarize the argument, scalable distributed object systems cannot be effective without interactions based on asynchronous method calls, with respect to mastering both complexity and efficiency. While acknowledged theories have been proposed for both asynchronous message passing (e.g., π-calculus) and objects (e.g., ς-calculus), no formal framework has been proposed for objects communicating solely with non-blocking method calls. This is exactly the ambition of the current book: to define a theory for distributed objects interacting with asynchronous method calls.

Starting from widely adopted object theory, the ς-calculus [3], a syntactically lightweight extension is proposed to take distribution into account. Two simple primitives are proposed: *Active* and *Serve*. The former turns an object into an independent and potentially remote activity; the latter allows such an active object to execute (serve) a pending remote call. On activation, an object becomes a remotely accessible entity with its own thread of control: *an active object*. In accordance with the above reasoning, we have chosen to make method calls to active objects systematically asynchronous. Synchronization is ensured with a natural dataflow principle: *wait-by-necessity*. An active object is blocked on the invocation of a not yet available result, i.e., a strict operation on an unknown future. A further level of asynchrony and low coupling is reached with the first-class nature of futures within wait-by-necessity; they can be passed between active objects as method parameters and returned as results.

The proposed calculus is named *Asynchronous Sequential Processes* (ASP), reflecting an important property: the sequentiality of active objects. Processes denote the potentially coarse-grain nature of active objects. Such processes are usually formed with a set of standard objects under the exclusive control of a root object. The proposed theory allows us to express a fundamental condition for *confluence*, alleviating for the programmer of the unscalable need to consider the interleaving of all instructions and communications. Furthermore, a property ensures *determinism*, stating that, whatever the order of communications, whatever the order of future updates, even in the presence of cycles, some systems converge towards a determinate global state. Apart from Process Networks [99, 100, 159], now close to 40 years old, few calculi

and languages ensure determinism, and even fewer in the context of stateful distributed objects interacting with asynchronous method calls. The potential of the proposed theory is further demonstrated by the capacity to cope with more advanced issues such as mobility, groups, and components.

One objective of the proposed theory is to be a *practical* one. Implementation strategies are covered. Several chapters explore a number of solutions, adapted to various settings (high-speed local area networks with buffer saving in mind, wide area networks with latency hiding as a primary goal, etc.), but each one still preserving semantics and properties. An illustration of such practicability is available under an open source Java API and environment, ProActive [134], which implements the proposed theory using a strategy designed to hide latency in the setting of wide area networks.

The first part of this book analyzes the issues at hand, reviewing existing languages and calculi.

Parts II and III formally introduce the proposed framework, defining the main properties of confluence and determinism.

Part IV reaches a new frontier and discusses issues at the cutting edge of software engineering, namely migration, reconfiguration, and component-based systems. From the proposed framework, we suggest a path that can lead to reconfigurable components. It demonstrates how we can go from asynchronous distributed objects to asynchronous distributed components, including collective remote method invocations (group communications), while retaining determinism.

With practicality in mind, Part V analyzes implementation issues, and suggests a number of strategies. We are aware that large-scale distributed systems encounter large variations in conditions, due to both localization in space and dynamic changes over time. Thus, potentially adaptive strategies for buffering and pipelining are proposed.

Finally, after a comparative evaluation of related formalisms, Part VI concludes and suggests directions for the future.

Acknowledgments

We are pleased to acknowledge discussions, collaboration, and joint work with many people as a crucial inspiration and contribution to the pages herein.

Without Isabelle Attali – Isa, *Project Leader of the OASIS team until December 26th 2004, this book would not be in your hands.*

Bernard Serpette significantly contributed to the development of the ASP calculus and related proofs.

All the other senior OASIS team members, Françoise Baude and Eric Madelaine, were very supportive and contributed in many aspects.

This book is also the result of many fruitful interactions between theory and practice. Many inspiring ideas came from the practical development of the ProActive library, and from contributors.

Special thanks go to Fabrice Huet and Julien Vayssière, the first two ProActive contributors, implementors, and testers. Many others recently had key contributions, especially the younger researchers and engineers in our team: Laurent Baduel, Tomás Barros, Rabéa Ameur-Boulifa, Javier Bustos, Arnaud Contes, Alexandre Di Costanzo, Christian Delbé, Felipe Luna Del Aguila, Matthieu Morel, and last, but not least, Romain Quilici.

Former Master's, Ph.D. students, engineers, were also a key source of maturation and inspiration: Alexandre Bergel, Roland Bertuli, Florian Doyon, Sidi Ould Ehmety, Alexandre Fau, Wilfried Klauser, Emmanuel Léty, Lionel Mestre, Olivier Nano, Arnaud Poizat, Yves Roudier, Marjorie Russo, David Sagnol.

Finally, colleagues from around the world, Martín Abadi, Gul Agha, Gérard Boudol, Luca Cardelli, David Crookall, Davide Sangiorgi, Akinori Yonezawa, and Andrew Wendelborn, made very useful comments on early drafts of the book or related research papers.

Nice - Sophia Antipolis
London
February 2005

Denis Caromel
Ludovic Henrio

Reading Paths and Teaching

Extra Material and Dependencies

You will find at the end of this book a list of notations and a summary of ASP syntax and semantics that should provide a convenient quick reference (Index of Notations, Syntax, Operational Semantics). This is followed by a graphical view of ASP properties (page 183), and the syntax of ASP extensions (Synchronizations, Migration, Groups, Components).

The Appendices detail formal definitions and proofs of the main theorems and properties introduced in Part III.

Figure 1 exhibits the dependencies between chapters and sections. Each chapter is best read after the preceding chapters. For example, in order to fully understand the group communication in ASP (Chap. 13), one should read Chaps 3, 4, Part III (Chaps. 6, 7, 8, 9), and Chap.10. Going down the lines (Fig. 1), one can follow the outcomes of chapters. For instance, still for group communication in Chap. 13, immediate benefits are parallel components (Sect. 14.5), and a practical implementation of typed group communication within ProActive (Chap. 16).

Text Book

Besides researchers and middleware designers, the material here can also be used as a text book for courses related to *models, calculi, languages for concurrency, parallelism, and distribution.* The focus is clearly on recent advances, especially object-orientation and asynchronous communications. Such courses can provide theoretical foundations, together with a perspective on practical programming and software engineering issues, such as distributed components.

The courses cover classical calculi such as CSP [88] and π-calculus [119, 120, 144], object-orientation using ς-calculus [3, 1, 2], and ASP [52], and advanced issues such as mobility, groups, and components. Overall, the objectives are threefold:

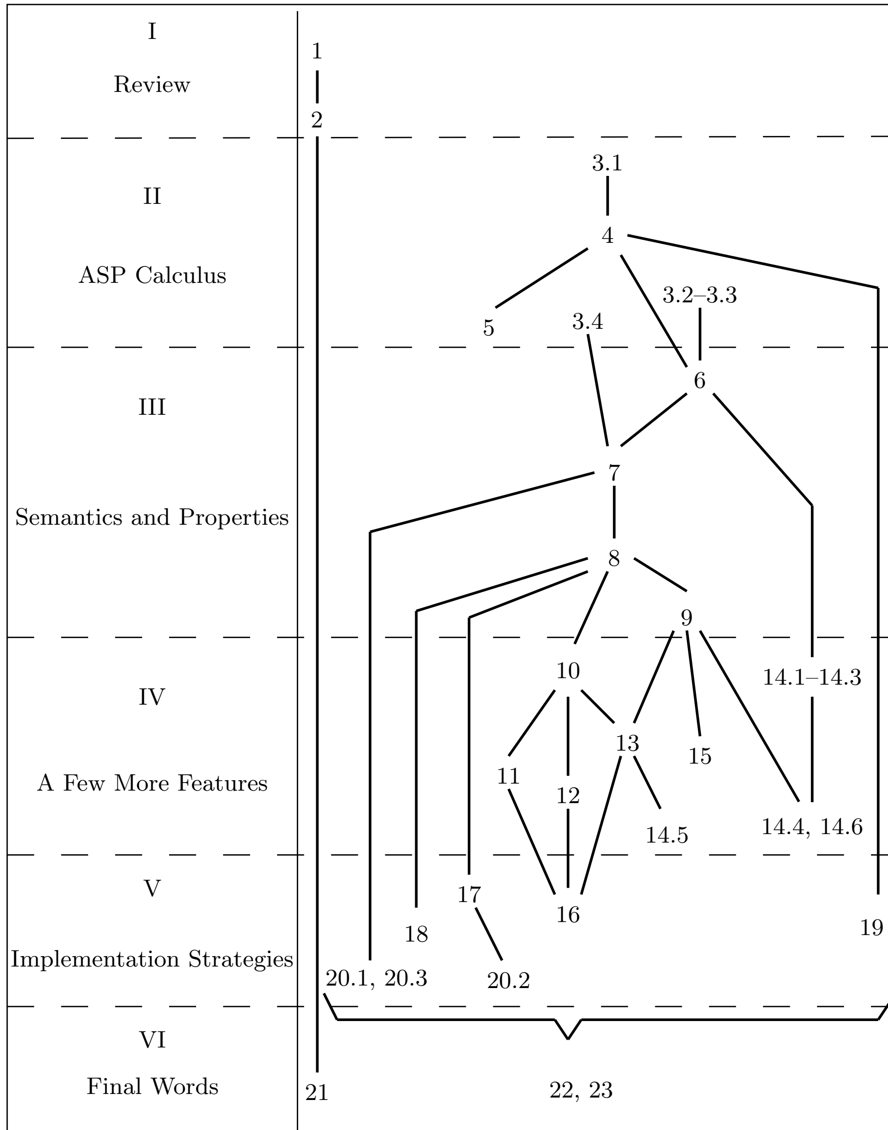


Fig. 1. Suggested reading paths

(1) study and analyze existing models of concurrency and distribution,
(2) survey their formal definitions within a few calculi,
(3) understand the implications on programming issues.

Depending on the objectives, the courses can be aimed at more theoretical aspects, up to proofs of convergence and determinacy within π-calculus and ASP, or targeted at more pragmatic grounds, up to practical programming

sessions using software such as PICT [132, 131] or *ProActive* [134].

Below is a suggested outline for a semester course, with references to online material, and chapters or sections of this book:

Models, Calculi, Languages for Concurrency, Parallelism, and Distribution

1.	Introduction to Distribution, Parallelism, Concurrency General Overview of Basic formalisms	1 [39]
2.	CCS, and/or Pi-Calculus	2.1.3 [73]
3.	Other Concurrent Calculi and Languages (Process Network, Multilist, Ambient, Join, ...)	2.1.4, 2.2 [125]
4.	Object-Oriented calculus: ς-calculus	2.1.5 [4]
5.	Overview of Concurrent Object Calculi (Actors, ABCL, Obliq and Øjeblik, $\pi o\beta\lambda$, **conc**ς-calculus, ...)	2.1.2, 2.3 [39]
6.	Asynchronous Method Calls and Wait-by-necessity ASP: Asynchronous Sequential Processes	3, 4, 5
7.	Semantics, Confluence, Determinacy	6, 7, 8, 9
8.	Advanced issues I: Confluent and non-confluent features, mobility	10, 11, 12 [125]
9.	Advanced issues II: Groups, Components	13, 14
10.	Open issue: reconfiguration Conclusion, Perspective, Wrap-up	15, 21, 22, 23

The Web page [39] gathers a broad range of information aimed at concurrent systems, also featuring parallel and distributed aspects. Valuable material for teaching models of concurrent computation, including CCS and π-calculus can be found at [73]. The Web page [4] is dedicated to the book *A Theory of Objects* [3]; it references pointers to courses using ς-calculus, some with teaching material available online. Finally, a comprehensive set of resources related to calculi for mobile processes is available at [125].

Assignments can include proofs of the confluence or non-confluence natures of a few features (e.g., delegation, explicit wait, method update, testing future or request reception, non-blocking services, join constructs, etc.). More practical assignments can involve designing and evaluating new future-update strategies, new request delivery protocols, or new schemes for pipelining control. Practicality can reach as far as implementing examples or prototypes, using PICT [132, 131], *ProActive* [134], or other programming frameworks.

A Theory of Distributed Objects online

We intend to maintain a Web page for general information, typos, etc. Extra material is also expected to be added (slides, exercises and assignments, contributions, reference to new related papers, etc.). This page is located at:

`http://www.inria.fr/oasis/caromel/TDO`

Do not hesitate to contact us to comment or to exchange information!

Part I

Review

1

Analysis

1.1 A Few Definitions

Definition 1.1 (Parallelism)
Execution of several activities or processes at the same time

Definition 1.2 (Concurrency)
Simultaneous access to a resource, physical or logical

Definition 1.3 (Distribution)
Several address spaces

Definition 1.4 (Asynchronous systems)
No global clock, and unbounded communication time

1.2 Distribution, Parallelism, Concurrency

1.2.1 Parallel Activities

Aspect **Activity** taking value in

- Process
- Expression evaluation
- Actor
- Active object

1.2.2 Sharing

Aspect **Sharing** taking value in

- Yes
- Some
- No

1.2.3 Communication

Aspect **Communication Base** taking value in
• Channel • RPC

Aspect **Communication Passing** taking value in
• Generalized reference • Copy, and deep copy • copy–restore • Lazy copy • Copy of activities (mobility)

Aspect **Communication Timing** taking value in
• Synchronous • Asynchronous with rendezvous • Asynchronous FIFO preserving • Asynchronous without guarantee

1.2.4 Synchronization

Definition 1.5 (Future)
A reference to a value unknown at creation, to be automatically filled up by some activity.
An automatic wait upon a strict operation on a future.

Aspect **Synchronization** taking value in
• Control • Filtering patterns (select, join, blocking service) • Dataflow • Future

1.2.5 Reactive vs. Proactive vs. Synchronous

Definition 1.6 (Reactive system)
A system is reactive if it always reacts quickly enough with respect to the occurrences of the stimuli and the dynamics of the environment.

Definition 1.7 (Synchrony hypothesis)

1. *Functional execution and communication time are both considered as null.*
2. *The entire system is placed under a unique global clock, usually a logical clock, defining global instants.*
   Reactivity occurs at each instant.

1.3 Objects

1.3.1 Object vs. Remote Reference and Communication

Aspect **Object RMI** taking value in

- Yes
- No

1.3.2 Object vs. Parallel Activity

Aspect **Object Activity** taking value in

- Yes, all objects (uniform)
- Yes, some objects (non-uniform)
- No

1.3.3 Object vs. Synchronization

Definition 1.8 (Wait-by-necessity)

Automatic and transparent creation of future objects upon remote method invocations.
Futures as generalized references passed between distributed activities.
Automatic wait upon strict operations on future objects.

Aspect **Wait-by-necessity** taking value in

- Yes
- No

1.4 Summary and Orientation

Aspects	**Possible Values:**
Activity	Process
	Expression evaluation
	Actor
	Active object
Sharing	Yes
	Some
	No
Communication Base	Channel
	RPC
Communication Passing	Generalized reference
	Copy and deep copy
	copy–restore
	Lazy copy
	Copy of activities (mobility)
Communication Timing	Synchronous
	Asynchronous with rendezvous
	Asynchronous FIFO preserving
	Asynchronous without guarantee
Synchronization	Control
	Filtering patterns (select, join, blocking service)
	Dataflow
	Future
Object RMI	Yes
	No
Object Activity	Yes, all objects (uniform)
	Yes, some objects (non-uniform)
	No
Wait-by-necessity	Yes
	No

Table 1.1. Aspects of distribution, parallelism, and concurrency

Aspects	**Values:**
Activity	Active object
Sharing	No
Communication Base	RPC
Communication Passing	Generalized reference to activities and futures
	Deep copy of objects
	Copy of activities (mobility)
Communication Timing	Asynchronous with rendezvous
Synchronization	Blocking service
	Future
Object RMI	Yes
Object Activity	Yes, some objects (non-uniform)
Wait-by-necessity	Yes

Table 1.2. Aspects of ASP

2

Formalisms and Distributed Calculi

2.1 Basic Formalisms

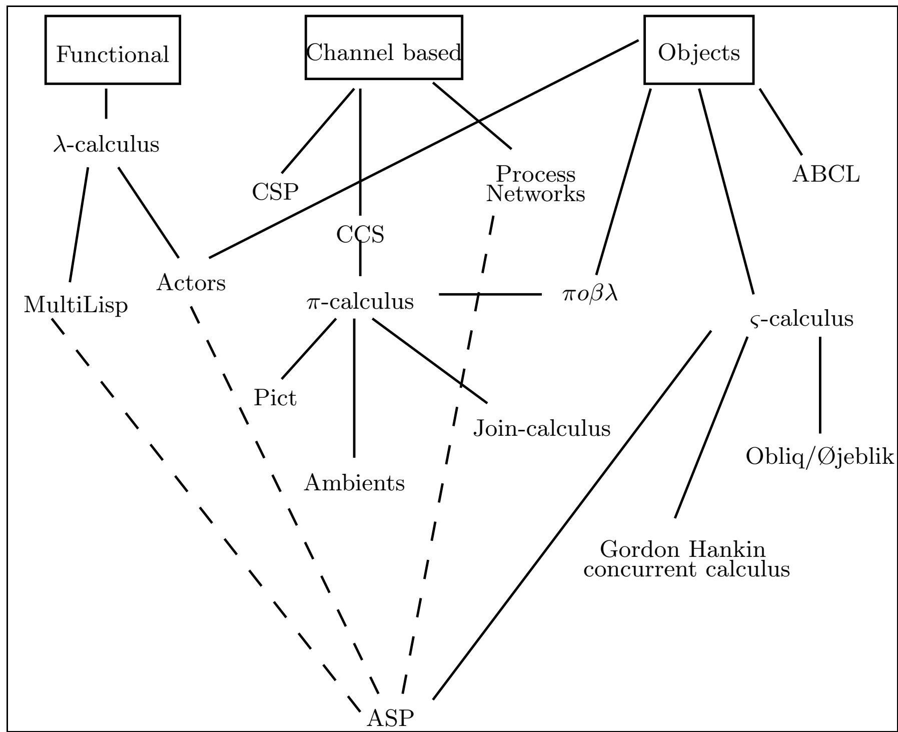


Fig. 2.1. Classification of calculi (informal)

2.1.1 Functional Programming and Parallel Evaluation

$$\begin{array}{rll} M, N ::= & x & \text{variable} \\ & \mid \lambda x.M & \text{abstraction} \\ & \mid (MN) & \text{application} \end{array}$$

```
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;

let rec member x btree =
  match btree with
        Empty -> false
      | Node(y, left, right) -> if x = y then true else
          if x < y then member x left else member x right;;

let rec insert x btree =
  match btree with
        Empty -> Node(x, Empty, Empty)
      | Node(y, left, right) ->
          if x <= y then
               Node(y, insert x left, right)
          else
              Node(y, left, insert x right);;
```

Fig. 2.2. A binary tree in CAML

2.1.2 Actors

```
⟨act program⟩ ::= ⟨behavior definition⟩* (⟨command⟩)

⟨behavior definition⟩ ::= (define (id {(with identifier ⟨pattern⟩)}*)
                                      ⟨communication handler⟩*)

⟨communication handler⟩ ::= (Is-communication⟨pattern⟩ do ⟨command⟩)

⟨command⟩ ::= let ⟨let binding⟩* ⟨command⟩
            | (if ⟨expression⟩ then ⟨command⟩ else ⟨command⟩)
            | (send ⟨expression⟩ ⟨expression⟩)
            | (become ⟨expression⟩)
```

Table 2.1. The syntax of an Actors language [9]

```
(define (Factorial ())
  (Is-Communication (a eval (with customer ≡ c)
                         (with number ≡ n)) do
      (become Factorial)
      (if (NOT (= n 0))
         (then (send x 1))
         (else (let (x (new FactCust (with customer c)
                              (with number n)))
                 (send Factorial (a eval (with customer x)
                              (with number n-1))))))))
(define (FactCust (with customer ≡ m)
                 (with number ≡ n))
   (Is-Communication (a number k) do
       (send m n*k)))
```

Fig. 2.3. A factorial actor [9]

Aspects	**Possible Values:**
Activity	Actor
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous, with a fairness guarantee
Synchronization	Filtering patterns (futures can exist at a higher level)
Object RMI	No
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.2. Aspects of Actors

2.1.3 π-calculus

$P, Q ::=$	$\mathbf{0}$	nil
	$\mid P \mid Q$	parallel composition
	$\mid (\nu x.P)$	restriction of name x
	$\mid \tau.P$	unobservable action
	$\mid x(y).P$	input
	$\mid x\langle y\rangle.Q$	output
	$\mid [x = y].Q$	name matching
	$\mid P + Q$	choice
	$\mid !P$	replication

Table 2.3. The syntax of π-calculus

$P \equiv Q$ if P is obtained from Q by change of bound names (alpha conversion)

$$P + \mathbf{0} \equiv P,\ P + Q \equiv Q + P,\ P + (R + R) \equiv (P + Q) + R$$

$$P|\mathbf{0} \equiv P,\ P|Q \equiv Q|P,\ P|(R|R) \equiv (P|Q)|R$$

$$(\nu x.(P|Q)) \equiv P|(\nu x.Q) \text{ if } x \notin fn(P),\ (\nu x.\mathbf{0}) \equiv \mathbf{0},\ (\nu y.(\nu x.P)) \equiv (\nu x.(\nu y.P))$$

$$!P \equiv P|!P$$

Table 2.4. π-calculus structural congruence

TAU:

$$\frac{}{\tau.P + M \to P}$$

REACT:

$$\frac{}{(x(y).P + M)|(x\langle z\rangle.Q + N) \to P\{\!\{y \leftarrow z\}\!\}|Q}$$

PAR:

$$\frac{P \to P'}{P|Q \to P'|Q}$$

RES:

$$\frac{P \to P'}{(\nu x.P) \to (\nu x.P')}$$

STRUCT:

$$\frac{P \to P' \wedge P \equiv Q \wedge P' \equiv Q'}{Q \to Q'}$$

Table 2.5. π-calculus reaction rules

Variants of π-calculus

Linear and Linearized Channels

What is Mobility?

2.1.4 Process Networks

2.1.5 ς-calculus

ι	store location (e.g., an integer)
$v ::= [m_i = \iota_i]^{i \in 1..n}$	result (m_i distinct)
$\sigma ::= \{\iota_i \mapsto \langle\varsigma(x_i)b_i, S_i\rangle\}^{i \in 1..n}$	store (ι_i distinct)
$S ::= \{x_i \mapsto v_i\}^{i \in 1..n}$	stack (x_i distinct)
$S \vdash \diamond$	well-formed store judgment
$\sigma \bullet S \vdash \diamond$	well-formed stack judgment
$\sigma \bullet S \vdash a \rightsquigarrow v \bullet \sigma'$	term reduction judgment

Aspects	Values:
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
	Copy of activities (mobility of activities) in HOπ
Communication Timing	Synchronous
Synchronization	Control
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.6. Aspects of π-calculus

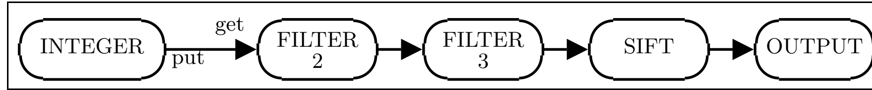


Fig. 2.4. Execution of the sieve of Eratosthenes in Process Networks

Aspects	Values:
Activity	Process
Sharing	No
Communication Base	Channel
Communication Passing	Copy
Communication Timing	Asynchronous FIFO preserving
Synchronization	Dataflow
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.7. Aspects of Process Networks

$a, b \in L ::= x$	variable
$\mid [m_j = \varsigma(x_i)a_i]^{i \in 1..n}$	object definition
$\mid a.m_i$	method invocation
$\mid a.l_i \Leftarrow \varsigma(x)b$	method update
$\mid clone(a)$	superficial copy
$\mid let\ x = a\ in\ b$	let

Table 2.8. The syntax of **impς**-calculus [3]

2.2 Concurrent Calculi and Languages

2.2.1 MultiLisp

2.2.2 PICT

2.2.3 Ambient Calculus

```
Process INTEGERS out QO
    Vars N; 1 → N;
    repeat INCREMENT N; PUT(N,QO) forever
Endprocess;

Process FILTER PRIME in QI out QO
    Vars N;
    repeat GET(QI) → N;
           if (N MOD PRIME)≠0 then PUT(N,QO) close
    forever
Endprocess;

Process SIFT in QI out QO
    Vars PRIME; GET(QI) → PRIME;
    PUT(PRIME,QO);  emit a discovered prime
    doco channels Q;
      FILTER(PRIME,QI,Q); SIFT(Q,QO);
    closeco
Endprocess;

Process OUTPUT in QI;
    repeat PRINT(GET(QI)) forever
Endprocess

Start doco channels Q1 Q2;
    INTEGERS(Q1);SIFT(Q1,Q2); OUTPUT(Q2);
    closeco;
```

Fig. 2.5. Sieve of Eratosthenes in Process Networks [100]

store $\emptyset$:

$$\frac{}{\emptyset \vdash \diamond}$$

store ι:

$$\frac{\sigma \bullet S \vdash \diamond \quad \iota \notin dom(\sigma)}{\{\iota \mapsto \langle \varsigma(x)b, S\rangle\} :: \sigma \vdash \diamond}$$

Table 2.9. Well-formed store

$$n[in\ m.P|Q]|m[R] \rightarrow m[n[P|Q]|r]$$
$$m[n[out\ m.P|Q]|r] \rightarrow n[P|Q]|m[R]$$
$$open\ n.P|n[Q] \rightarrow P|Q$$
$$(x).P|\langle M\rangle \rightarrow P\{x \leftarrow M\}$$

stack $\emptyset$:

$$\frac{\sigma \vdash \diamond}{\sigma \bullet \emptyset \vdash \diamond}$$

stack x:

$$\frac{\sigma \bullet S \vdash \diamond \quad \iota \notin dom(\sigma)}{\sigma \bullet \left(\{x \mapsto [m_i = \iota_i]^{i \in 1..n}\} :: S\right) \vdash \diamond}$$

Table 2.10. Well-formed stack

x:

$$\frac{\sigma \cdot (S :: \{x \mapsto v\} :: S') \vdash \diamond}{\sigma \bullet (S :: \{x \mapsto v\} :: S') \vdash x \rightsquigarrow v \bullet \sigma}$$

object:

$$\frac{\sigma \bullet S \vdash \diamond \quad \forall i \in i..n,\ \iota_i \notin dom(\sigma)}{\sigma \bullet S \vdash [m_j = \varsigma(x_i)a_i]^{i \in 1..n} \rightsquigarrow [m_j = \iota_i]^{i \in 1..n} \bullet \{\iota_i \mapsto \langle \varsigma(x_i)a_i, S\rangle^{i \in 1..n}\} :: \sigma}$$

select:

$$\frac{\begin{array}{c}\sigma \bullet S \vdash a \rightsquigarrow [m_i = \iota_i]^{i \in 1..n} \bullet \sigma' \\ \sigma'(\iota_j) = \langle \varsigma(x_j)a_j, S'\rangle \quad x_j \notin dom(S') \quad j \in 1..n \\ \sigma' \bullet (x_j \mapsto [m_j = \iota_i]^{i \in 1..n} :: S') \vdash a_j \rightsquigarrow v \bullet \sigma''\end{array}}{\sigma \bullet S \vdash a.m_j \rightsquigarrow v \bullet \sigma''}$$

update:

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [m_i = \iota_i]^{i \in 1..n} \bullet \sigma' \quad j \in 1..n \quad \iota_j \in dom(\sigma')}{\sigma \bullet S \vdash a.m_j \Leftarrow \varsigma(x)b \rightsquigarrow [m_i = \iota_i]^{i \in 1..n} \bullet (\{\iota_j \mapsto \langle \varsigma(x)b, S\rangle\} + \sigma')}$$

clone:

$$\frac{\sigma \bullet S \vdash a \rightsquigarrow [m_i = \iota_i]^{i \in 1..n} \bullet \sigma' \quad \forall i \in 1..n,\ \iota_i \in dom(\sigma') \wedge \iota'_i \notin dom(\sigma')}{\sigma \bullet S \vdash clone(a) \rightsquigarrow [m_i = \iota'_i]^{i \in 1..n} \bullet (\{\iota'_i \mapsto \sigma'(\iota_i)\} + \sigma')}$$

let:

$$\frac{\iota\sigma \bullet S \vdash a \rightsquigarrow v' \bullet \sigma' \quad \sigma' \bullet (\{x \mapsto v'\}) :: S \vdash b \rightsquigarrow v'' \bullet \sigma''}{\sigma \bullet S \vdash let\ x = a\ in\ b \rightsquigarrow v'' \bullet \sigma''}$$

Table 2.11. Semantics of **imp**ς-calculus (big-step, closure based)

2.2.4 Join-calculus

2.2.5 Other Expressions of Concurrency

CML

Kell-calculus

Steele Shared Memory Non-interference

Montanari Tile-Based Semantics

Functional Nets

2.3 Concurrent Object Calculi and Languages

2.3.1 ABCL

```
Sieve ≜ [m = ς(s) λ(n)
                let sieve' = clone(s)
                in   s.prime := n;
                     s.next := sieve';
                     s.m⇐ς(s') λ(n')
                            if(n' mod n) = 0
                            then [ ]
                            else sieve'.m(n');
                     [ ],
         prime = ς(x) x.prime,
         next = ς(x) x.next];
```

⇐ denotes the method update: modifies the body of a method.

The sieve can be used in the following way:

for i in 2..99 *do sieve.m(i)* initializes primes ≤ 100

sieve.next.next.prime returns the third prime

Fig. 2.6. Sieve of Eratosthenes in ς-calculus [3]

```
binClass ≜ [new =
              ς(z) [isleaf = ς(s) z.(isleaf s),
                    lft = ς(s) z.(lft s),
                    rht = ς(s) z.(rht s),
                    consLft = ς(s) z.(consLft s),
                    consRht = ς(s) z.(consRht s)],
           isLeaf = λ(self) true,
           lft = λ(self) self.lft
           rht = λ(self) self.rht
           consLft = λ(self) λ(newlft)
                       ((self.isleaf := false).lft := newlft).rht := self,
           consRht = λ(self) λ(newrht)
                       ((self.isleaf := false).lft := self).rht := newrht]
```

Fig. 2.7. Binary tree in ς-calculus [3]

Aspects	**Values:**
Activity	Expression evaluation
Sharing	Yes
Communication Base	No communication
Communication Passing	No communication
Communication Timing	No communication
Synchronization	Future
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.12. Aspects of MultiLisp

$Val ::=$	Id	Variable
	$[\ Label\ Val\ \ldots\ Label\ Val\ ]$	Record
	$\{\ Type\ \}\ Val$	Polymorphic package
	$(\ rec\ :\ \mathtt{T}\ Val\ )$	Rectype value
	$String$	String constant
	$Char$	Character constant
	$Bool$	Boolean constant
$Label ::=$	$\langle empty \rangle$	Anonymous label
	$Id\ =$	Explicit label
$Pat ::=$	$Id\ :\ Type$	Variable pattern
	$_\ :\ Type$	Wildcard pattern
	$Id\ :\ Type\ @\ Pat$	Layered pattern
	$[\ Label\ Pat\ \ldots\ Label\ Pat\ ]$	Record pattern
	$\{\ Id\ <\ Type\ \}\ Pat$	Package pattern
	$(\ rec\ :\ \mathtt{T}\ Pat\ )$	Rectype pattern
$Abs ::=$	$Pat\ =\ Proc$	Process abstraction
$Proc ::=$	$Val\ !\ Val$	Output atom
	$Val\ ?\ Abs$	Input prefix
	$Val\ ?*\ Abs$	Replicated input prefix
	$(\ Proc \mid Proc\ )$	Parallel composition
	$(Dec\ Proc\ )$	Local declaration
	$\mathtt{if}\ Val\ \mathtt{then}\ Proc\ \mathtt{else}\ Proc$	Conditional

Table 2.13. A syntax for PICT [132]

```
def fib[n:Int r:!Int] =
  if (|| (== n 0) (== n 1)) then
     r!1
  else
    r!(+ (fib (- n 1)) (fib (- n 2)))

run printi!(fib 7)
```

Fig. 2.8. A simple Fibonacci example in PICT [130]

```
run
(def fact [n:Int r:!Int]=
   (new br:^Bool
      ( {- calculate n=0 -}
        ==![n0 (rchan br)]
      | {- is n=0? -}
        br?b =
          if b then
            {- yes: return 1 as result -}
            r!1
          else
            {- no ... -}
          (new nr:^Int
            ( {- subtract one from n -}
              -![n 1 (rchan nr)]
            | nr?nMinus1 =
              {- make a recursive call to compute fact(n-1) -}
              (new fr?f =
                ( fact!nMinus1 fr]
                | fr?f =
                    {- multiply n by fact(n-1) and send the result
                       on the original result channel r -}
                    *![f n (rchan r)]
                ))))))
  new r:^Int
   ( fact![5 r]
   | r?f = printi!f )
```

Fig. 2.9. A factorial example in the core PICT language [130]

$$acquire\ n.P \triangleq open\ n.P$$
$$release\ n.P \triangleq n[\ ]|P$$

Fig. 2.10. Locks in ambients [47]

Aspects	Values:
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous without guarantee
Synchronization	Filtering patterns (pattern matching)
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.14. Aspects of PICT

$P, Q ::=$	$(\nu n)P$	restriction
	$\mathbf{0}$	inactivity
	$P \vert Q$	composition
	$!P$	replication
	$n[P]$	ambient
	$M.P$	action
$M ::=$	$in\ n$	can enter n
	$out\ n$	can exit n
	$open\ n$	can open n

Table 2.15. The syntax of Ambient calculus

$buf\ n$	$\triangleq$	$n[!open\ io]$	a channel buffer
$(ch\ n)$	$\triangleq$	$(\nu n)\,(buf\ n \mid P)$	a new channel
$n(x).P$	$\triangleq$	$(\nu p)\,(io[in\ n.(x).p[out\ n.P]] \mid open\ p)$	channel input
$n\langle M\rangle$	$\triangleq$	$io[in\ n.\langle M\rangle]$	async channel output

Fig. 2.11. Channels in ambients [47]

Aspects	Values:
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference, with local effect
	Copy of ambient (mobility)
Communication Timing	Synchronous
Synchronization	Control
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.16. Aspects of Ambients

$P ::= \mid x\langle\tilde{y}\rangle$	message emission
$\mid \mathbf{def}\ D\ \mathbf{in}\ P$	definition of ports
$\mid P \vert P$	parallel composition
$\mid \mathbf{0}$	null process
$D ::= J \triangleright P$	rule matching join pattern J (trigger)
$\mid D \wedge D$	connection of rules
$\mid \mathbf{T}$	empty definition
$J ::= x\langle\tilde{y}\rangle$	message pattern
$\mid J \vert J$	joined patterns

Table 2.17. The syntax of the join-calculus

$\vdash P \vert P' \leftrightarrow \vdash P, P'$	
$\vdash \mathbf{0} \leftrightarrow \vdash$	
$\mathbf{T} \vdash \leftrightarrow \vdash$	
$\vdash \mathbf{def}\ D\ \mathbf{in}\ P \leftrightarrow D\sigma \vdash P\sigma$	σ creates fresh channels
$J \triangleright P \vdash J\sigma \longrightarrow J \triangleright P \vdash P\sigma$	

Table 2.18. Main rules defining evaluation in the Join calculus

$$\mathbf{def}\ mkcell\langle v_0, \kappa_0\rangle \triangleq \begin{pmatrix} \mathbf{def} & get\langle\kappa\rangle \vert s\langle v\rangle \triangleright \kappa\langle v\rangle \vert s\langle v\rangle \\ \wedge & set\langle u, \kappa\rangle \vert s\langle v\rangle \triangleright \kappa\langle\rangle \vert s\langle u\rangle \\ \mathbf{in} & s\langle v_0\rangle \vert \kappa_0\langle get, set\rangle \end{pmatrix}$$

Fig. 2.12. A cell in the join-calculus [68]

Aspects	**Values:**
Activity	Expression evaluation
Sharing	No
Communication Base	Channel
Communication Passing	Generalized reference
Communication Timing	Asynchronous without guarantee
Synchronization	Filtering patterns (join)
Object RMI	No
Object Activity	No
Wait-by-necessity	No

Table 2.19. Aspects of the Join-Calculus

```
[object Buffet
   (state declare-the-buffer-state )
   (script

      (=> [:put elt]     ; Put an element in the buffer
        (if full
           then (select   ; Waits for a [:get] message
                   (=>[:get] remove-from-storage-and-return )
                 )
         )
         store-elt
      )

      (=> [:get]         ; Get an element from the buffer
        (if empty
           then (select   ; Waits for a [:put ...] message
                   (=>[:put elt] send-elt-to-get-caller )
                  )
           else remove-from-buffer-send-it-to-get-caller
         )
      )
   )
]
```

Fig. 2.13. Bounded buffer in ABCL [161]

```
    ; Synchronous communication:
    ; send and wait for message execution
    ; Originally called:  Now Type Message Passing
x := [T  <== M]
        ; Current activity send a message M to T
        ; wait for message execution and return value,
        ; which is stored in x

    ; Asynchronous one-way communication:
    ; send and does not wait for
    ; message execution, no reply
    ; Originally called:   Past Type Message Passing
[T  <= M]
        ; Current activity send a message M to T
        ; no wait

    ; Asynchronous communication with explicit future:
    ; send and does not wait for message execution,
    ; reply to be sent later in  x
    ; Originally called:  Future Type Message Passing
[T  <= M $ x]
      ; Current activity send a meessage M to T
      ; non-blocking, the result will be put
      ; asynchronously in x
...
(ready? x)
      ; Test if x is still awaited
```

Fig. 2.14. The three communication types in ABCL

2.3.2 Obliq and Øjeblik

Migration

2.3.3 The $\pi o\beta\lambda$ Language

2.3.4 Gordon and Hankin Concurrent Calculus: concς-calculus

2.4 Synthesis and Classification

Aspects	Values:
Activity	Active object
Sharing	No
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous, and
	Asynchronous FIFO preserving
Synchronization	Control
	Filtering patterns (select)
	Future
Object RMI	Yes
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.20. Aspects of ABCL

```
let sieve =
{ m =>
   meth(s, n)
     print(n);
     let s0 = clone(s);
     s.m := meth(s1,n1)
              if (n1 % n) is 0 then ok
              else s0.m(n1)
              end
            end;
     end};

 print the primes <100
for i=2 to 100 do sieve.m(i) end;
```

Fig. 2.15. Prime number sieve in Obliq

$a, b \in L ::=$	s, x, y	variable
	$\mid [m_j = \varsigma(s_i, \tilde{x}_j)a_i]^{i \in 1..n}$	object definition
	$\mid a.m_i\langle\tilde{b}\rangle$	method invocation
	$\mid a.l_i \Leftarrow \varsigma(s, \tilde{x})b$	method update
	$\mid a.clone$	superficial copy
	$\mid a.alias\langle b\rangle$	object aliasing
	$\mid a.surrogate$	object surrogation
	$\mid a.ping$	object identity
	$\mid let\ x = a\ in\ b$	let
	$\mid fork\langle a\rangle$	thread creation
	$\mid join\langle a\rangle$	thread destruction

Table 2.21. The syntax of Øjeblik

Aspects	**Values:**
Activity	Process
Sharing	Yes
Communication Base	RPC
Communication Passing	Generalized reference
	Copy of activity (mobility) as cloning+aliasing
Communication Timing	Synchronous
Synchronization	Control
Object RMI	Yes
Object Activity	No
Wait-by-necessity	No

Table 2.22. Aspects of Obliq and Øjeblik

```
class TO
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W);
    return

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then return L!Search(X)
              else return R!Search(X)
```

Fig. 2.16. Binary tree in (a language inspired by) $\pi o\beta\lambda$ [110]

Aspects	**Values:**
Activity	Active object
Sharing	No
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous with early return
Synchronization	Control
Object RMI	Yes
Object Activity	Yes, all objects (uniform)
Wait-by-necessity	No

Table 2.23. Aspects of $\pi o\beta\lambda$

```
class TO
var K:NAT, V:ref(A), L:ref(T), R:ref(T)

method Insert(X:NAT, W:ref(A))
    return ;
    if K=nil then (K:=X ; V:=W ; L:=new(T) ; R:=new(T))
    else if X=K then V:=W
         else if X<K then L!Insert(X,W)
              else R!Insert(X,W)

method Search(X:NAT):ref(A)
    if K=nil then return nil
    else if X=K then return V
         else if X<K then commit L!Search(X)
              else commit R!Search(X)
```

Fig. 2.17. $\pi o \beta \lambda$ parallel binary tree, equivalent to Fig. 2.16 [110]

Results:		
$u, v \in L ::= x$		variable
	$\mid p$	name
Denotations:		
$d ::= [m_j = \varsigma(x_i)a_i]^{i \in 1..n}$		object
Terms:		
$a, b \in L ::= u$		result
	$\mid p \mapsto d$	denomination
	$\mid u.m_i$	method invocation
	$\mid u.l_i \Leftarrow \varsigma(x)b$	method update
	$\mid clone(u)$	superficial copy
	$\mid let\ x = a\ in\ b$	let
	$\mid a \upharpoonright b$	parallel composition
	$\mid (\nu p)a$	restriction

Table 2.24. The syntax of **conc**ς-calculus [78]

Aspects	**Values:**
Activity	Process
Sharing	Yes
Communication Base	RPC
Communication Passing	Generalized reference
Communication Timing	Synchronous
Synchronization	Control
Object RMI	Yes
Object Activity	No
Wait-by-necessity	No

Table 2.25. Aspects of **conc**ς-calculus

		Functional		Channel Based		Object	
Languages **Aspects**	**ASP**	**Actors**	**MultiLisp**	**π-calculus**	**Process Networks**	**$\pi o\beta\lambda$**	**Obliq** **Øjeblik**
Activity	Active object	Actor	Exp.	Exp.	Process	Active object	Process
Sharing	No	No	Yes	No	No	No	Yes
Communication Base	RPC	Channel	No com.	Channel	Channel	RPC	RPC
Communication Passing	GR:activities+futures Deep copy of objects Copy of activities (mobility)	GR	No com.	GR	GR only in reconfiguration Copy	GR	GR Copy of Activity (mobility as cloning+aliasing)
Communication Timing	Asynchronous with rendezvous	Asynchronous (fairness)	No com.	Synchronous	Asynchronous FIFO preserving	Synchronous with early return	Synchronous
Synchronization	Blocking service Future	Filtering Patterns	Future	Control	Dataflow Blocking service	Control	Control
Object RMI	Yes	No	No	No	No	Yes	Yes
Object Activity	Yes, non-uniform	Yes, uniform	No	No	No	Yes, uniform	No
Wait-by-necessity	Yes	No	No	No	No	No	No

Exp.= Expression evaluation   No com.= No communication   GR = Generalized Reference

Table 2.26. Summary of a few calculi and languages

Part II

ASP Calculus

3

An Imperative Sequential Calculus

3.1 Syntax

$a, b \in L ::= x$	variable
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition
$\mid a.l_i$	field access
$\mid a.l_i := b$	field update
$\mid a.m_j(b)$	method call
$\mid clone(a)$	superficial copy
$\mid \iota$	location (not in source terms)

Table 3.1. Syntax of ASP sequential calculus

$$\begin{aligned} Point \triangleq\ & [x = 0,\ y = 0,\ color = [R = 0, G = 0, B = 0; print = \ldots]; \\ & getX = \varsigma(s,p)s.x,\ setX = \varsigma(s,p)s.x := p, getColor = \varsigma(s,p)s.color, \ldots] \end{aligned}$$

$$let\ x = a\ in\ b \triangleq [; m = \varsigma(z, x)b].m(a)$$

$$a; b \triangleq [; m = \varsigma(z, z')b].m(a)$$

$$\begin{aligned} \lambda x.b &\triangleq [arg = [], val = \varsigma(x, y)b\{\!\{x \leftarrow x.arg\}\!\}] \\ (b\ a) &\triangleq (clone(b).arg := a).val([]) \end{aligned}$$

$$\begin{aligned} let\ x = a\ and\ y = b\ in\ c \triangleq\ & let\ o = [x = [], y = []]\ in \\ & let\ x = a\{\!\{x \leftarrow o.x, y \leftarrow o.y\}\!\}\ in \\ & let\ y = b\{\!\{x \leftarrow o.x, y \leftarrow o.y\}\!\}\ in \\ & o.x := x; o.y := y; c \end{aligned}$$

$$let\ x \ = \ [f = y] \ \ and\ y = [g = x] \ \ in\ \ \ldots$$

3.2 Semantic Structures

3.2.1 Substitution

$\theta ::= \{\!\!\{ b \leftarrow c \}\!\!\}$.

3.2.2 Store

$$o ::= [l_i = \iota_i ; m_j = \varsigma(x_j, y_j) a_j]^{i \in 1..n}_{j \in 1..m}$$

$$\sigma ::= \{\iota_i \mapsto o_i\}$$

$$(\sigma + \sigma')(\iota) = \begin{array}{ll} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{array}$$

3.2.3 Configuration

Definition 3.1 (Well-formed sequential configuration)

$$\vdash (a, \sigma)\ \text{OK} \Leftrightarrow \begin{cases} locs(a) \subseteq dom(\sigma) \wedge fv(a) = \emptyset \\ \forall \iota \in dom(\sigma),\ locs(\sigma(\iota)) \subseteq dom(\sigma) \ \wedge \ fv(\sigma(\iota)) = \emptyset \end{cases}$$

Definition 3.2 (Equivalence on sequential configurations)

$$(a, \sigma) \equiv (a', \sigma') \Leftrightarrow \exists \theta,\ (a\theta, \sigma\theta) = (a', \sigma')$$

3.3 Reduction

$$\begin{array}{l} \mathcal{R} ::= \bullet \,|\, [l_i = \iota_i, l_k = \mathcal{R}, l_{k'} = b_{k'} ; m_j = \varsigma(x_j, y_j) a_j]^{i \in 1..k-1, k' \in k+1..n}_{j \in 1..m} \\ \quad |\, \mathcal{R}.m_i \,|\, \mathcal{R}.m_j(b) \,|\, \iota.m_j(\mathcal{R}) \,|\, \mathcal{R}.l_i := b \,|\, \iota.l := \mathcal{R} \,|\, clone(\mathcal{R}) \end{array}$$

$$\mathcal{R}[a] = \mathcal{R}\{\!\!\{ \bullet \leftarrow a \}\!\!\}$$

3.4 Properties

Property 3.3 (Well-formed sequential reduction)

$$\vdash (a, \sigma)\ \text{OK} \ \wedge \ (a, \sigma) \rightarrow_S (b, \sigma') \quad \Longrightarrow \quad \vdash (b, \sigma')\ \text{OK}$$

Sequential Determinism

Property 3.4 (Determinism)

$$c \rightarrow_S d \ \wedge \ c \rightarrow_S d' \Rightarrow \ d \equiv d'$$

STOREALLOC:

$$\frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \mapsto o\} :: \sigma)}$$

FIELD:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n}{(\mathcal{R}[\iota.l_k], \sigma) \rightarrow_S (\mathcal{R}[\iota_k], \sigma)}$$

INVOKE:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..m}{(\mathcal{R}[\iota.m_k(\iota')], \sigma) \rightarrow_S (\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}], \sigma)}$$

UPDATE:

$$\frac{\begin{array}{c}\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n \\ o' = [l_i = \iota_i, l_k = \iota', l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..k-1,\, k' \in k+1..n}_{j \in 1..m}\end{array}}{(\mathcal{R}[\iota.l_k := \iota'], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \rightarrow o'\} + \sigma)}$$

CLONE:

$$\frac{\iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\iota'], \{\iota' \mapsto \sigma(\iota)\} :: \sigma)}$$

Table 3.2. Sequential reduction

4

Asynchronous Sequential Processes

4.1 Principles

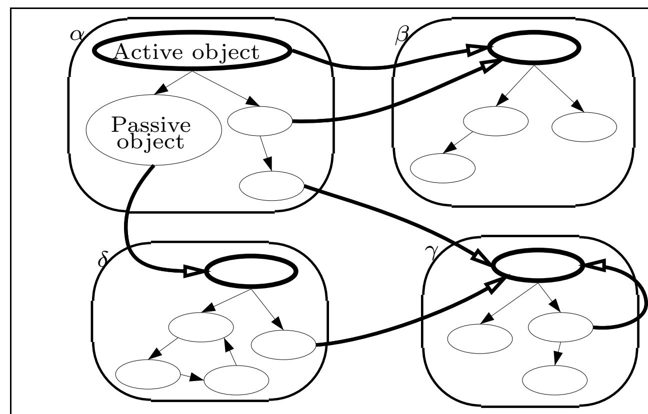


Fig. 4.1. Objects and activities topology

$$let\ p = Active(Point, \emptyset)\ in\ let\ col = p.getColor()\ in\ p.setX(2); col.print(),$$

4.2 New Syntax

$$P, Q ::= \alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[\ldots] \parallel \ldots$$

$a, b \in L ::= ...$		
	$\vert Active(a, m_j)$	Activates object: deep copy + activity creation, m_j is the activity method or $\emptyset$ for FIFO service
	$\vert Serve(M)$	Serves a request among a set of method labels,
	$\vert a \Uparrow f, b$	a with continuation b (not in source terms)

where M is a set of method labels used to specify the request that has to be served:

$$M = m_{k_1}, \ldots, m_{k_h}$$

Table 4.1. Syntax of ASP parallel primitives

$a, b \in L ::= x$	variable,
$\vert\, [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition
$\vert\, a.l_i$	field access
$\vert\, a.l_i := b$	field update
$\vert\, a.m_j(b)$	method call
$\vert\, clone(a)$	superficial copy
$\vert Active(a, m_j)$	activity creation
$\vert Serve(m_{k_j})^{j \in 1..h}$	service primitive

Table 4.2. Syntax of ASP calculus

4.3 Informal Semantics

4.3.1 Activities

4.3.2 Requests

4.3.3 Futures

4.3.4 Serving Requests

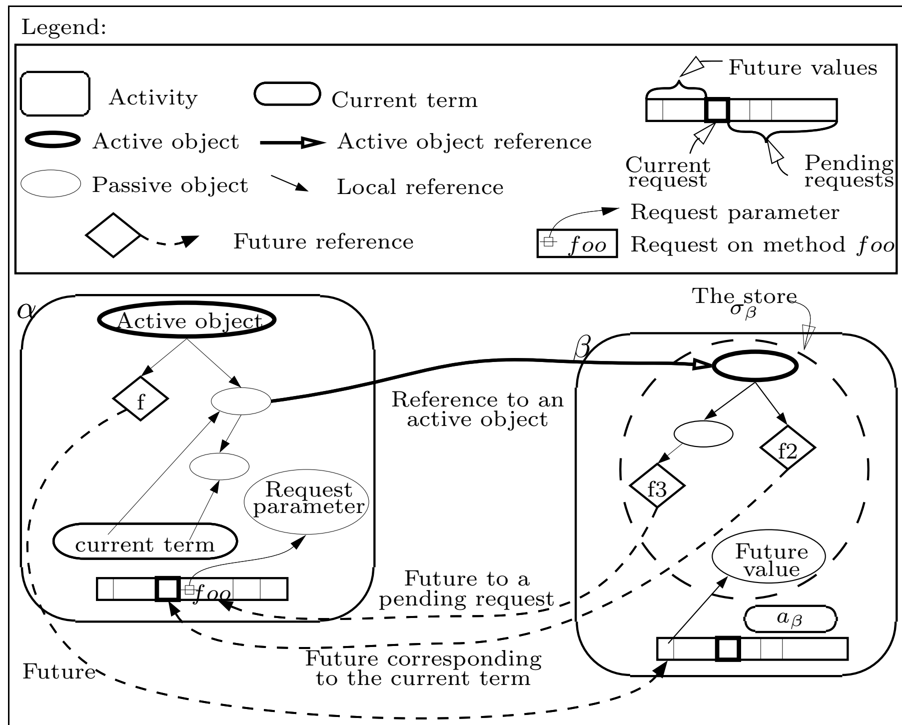


Fig. 4.2. Example of a parallel configuration

5

A Few Examples

$$
\begin{array}{ll}
\mathit{let}\ x = \mathit{Active}(a, s_1) & \triangleq \mathit{let}\ \mathit{srv}\ =\ \mathit{Active}([x = [], y = [];\\
\mathit{and}\ y = \mathit{Active}(b, s_2)\ \mathit{in}\ c & \qquad getX = \varsigma(s)s.x,\\
& \qquad getY = \varsigma(s)s.y,\\
& \qquad setX = \varsigma(s, x, y)(s.x := \mathit{Active}(a, s_1); s.x),\\
& \qquad setY = \varsigma(s, x, y)(s.y := \mathit{Active}(b, s_2); s.y),\\
& \qquad service = \varsigma(s)Serve(setX); Serve(setY);\\
& \qquad\qquad Repeat(Serve(getX); Serve(getY))\\
& \qquad], service)\\
& \mathit{in}\\
& [; m = \varsigma(s, x, y)c].m(\\
& \qquad\qquad srv.setX(srv.getX(), srv.getY()),\\
& \qquad\qquad srv.setY(srv.getX(), srv.getY()))
\end{array}
$$

5.1 Binary Tree

$$
\begin{array}{l}
\mathit{let}\ tree = (BT.new).add(3, 4).add(2, 3).add(5, 6).add(7, 8)\mathit{in}\\
\qquad [a = tree.search(5), b = tree.search(3)].b := tree.search(7)
\end{array}
$$

5.2 Distributed Sieve of Eratosthenes

Another Formulation

5.3 From Process Networks to ASP

5.4 Example: Fibonacci Numbers

5.5 A Bank Account Server

$$
\begin{array}{l}
BT \triangleq [new = \varsigma(c)[empty = true, left = [], right = [], key = [], value = [],\\
\quad\quad\quad search = \varsigma(s,k)(c.search\, s\, k), add = \varsigma(s,k,v)(c.add\ s\, k\, v)],\\
\quad search = \varsigma(c)\lambda s\, k.if\ (s.empty)\ then\ []\\
\quad\quad\quad else\ if\ (s.key == k)\ then\ s.value\\
\quad\quad\quad else\ if\ (s.key > k)\ then\ s.left.search(k)\\
\quad\quad\quad\quad else\ s.right.search(k),\\
\quad add = \varsigma(c)\lambda s\, k\, v.if\ (s.empty)\ then(s.right := Factory(s);\\
\quad\quad\quad\quad\quad s.left := Factory(s); s.value := v;\\
\quad\quad\quad\quad\quad s.key := k; s.empty := false;\ s)\\
\quad\quad else\ if\ (s.key > k)\ then\ s.left.add(k,v)\\
\quad\quad else\ if\ (s.key < k)\ then\ s.right.add(k,v)\\
\quad\quad else\ s.value := v;\ s \quad]
\end{array}
$$

where: $Factory(s) \triangleq s.new$ in the sequential case and
$Factory(s) \triangleq Active(s.new, \emptyset)$ for the concurrent binary tree.

Fig. 5.1. Example: a binary tree

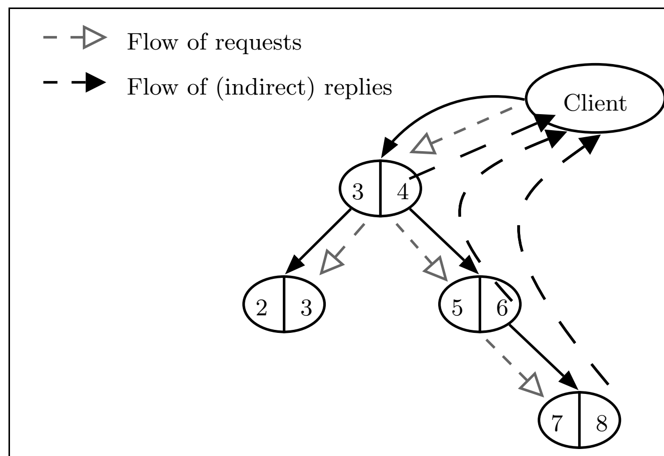


Fig. 5.2. Topology and communications in the parallel binary tree

$$
\begin{array}{l}
let\ Integer = Active([n = 1; get = \varsigma(s)(s.n := s.n + 1; s.n)], \emptyset)\ in \\
let\ Sieve = [parent = [], prime = 0; init = \varsigma(s, par)s.parent := par, \\
\qquad get = \varsigma(s) let\ n = parent.get()\ in \\
\qquad\qquad if (n \bmod s.prime \neq 0)\ then\ n\ else\ s.get()]\ in \\
let\ Sift = [source = Integer; \\
\qquad act = \varsigma(s) Repeat(let\ n = s.source.get()\ in \\
\qquad\qquad print(n); Sieve.prime := n; \\
\qquad\qquad s.source := Active(clone(Sieve.init(s.source)), \emptyset))]\ in \\
Active(Sift, act)
\end{array}
$$

Fig. 5.3. Example: sieve of Eratosthenes (pull)

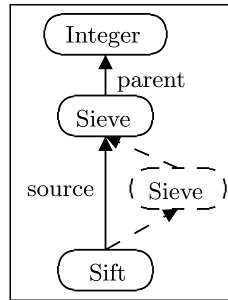


Fig. 5.4. Topology of sieve of Eratosthenes (pull)

$$
\begin{array}{l}
let\ Sieve = [N = 0, prime = 0; next = []; put = \varsigma(s, n)s.N := n, \\
\quad act = \varsigma(s) Serve(put); Display.put(s.N); \\
\qquad\qquad s.prime := s.N; s.next := Active(s, act); \\
\qquad\qquad Repeat(Serve(put)); \\
\qquad\qquad if\ (s.N \bmod s.prime \neq 0)\ then\ s.next.put(s.N))]\ in \\
let\ Integer = [n = 1; first = Active(Sieve, act); \\
\qquad act = \varsigma(s) Repeat(s.n := s.n + 1; s.first.put(s.n))]\ in \\
\quad Active(Integer, act)
\end{array}
$$

where *Display* is an object collecting and printing the prime numbers.

Fig. 5.5. Example: sieve of Eratosthenes (push)

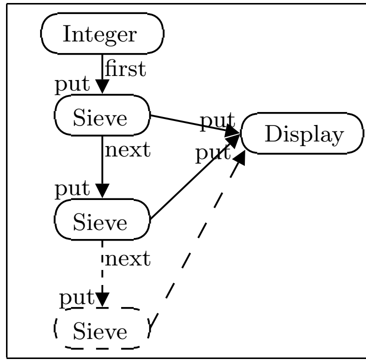


Fig. 5.6. Topology of sieve of Eratosthenes (push)

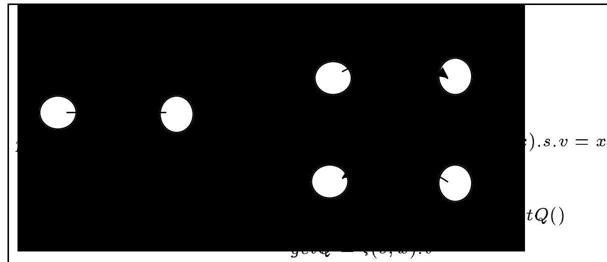


Fig. 5.7. Process Network vs. object network

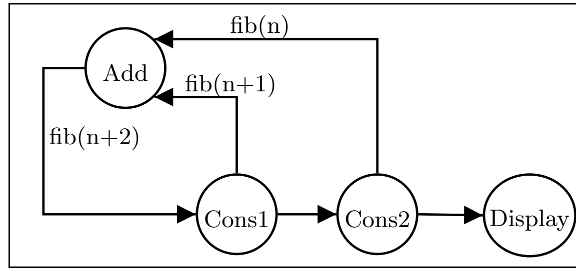


Fig. 5.8. Fibonacci number processes

$$
\begin{array}{l}
let\ Add = Active([n1 = 0, n2 = 0;\\
\quad service = \varsigma(s)\\
\quad\quad Repeat(Serve(set1); Serve(set2); Cons1.snd(s.n1 + s.n2)),\\
\quad set1 = \varsigma(s,n)s.n1 := n, set2 = \varsigma(s,n)s.n2 := n], service)\\
and\ Cons1 = Active(\\
\quad [; service = \varsigma(s)\ (Add.set1(1); Cons2.snd(1); Repeat(Serve(snd)))\\
\quad\ snd = \varsigma(s,n)(Add.set1(n); Cons2.snd(n))], service)\\
and\ Cons2 = Active(\\
\quad [; service = \varsigma(s)\ (Add.set2(0); Display.snd(0); Repeat(Serve(snd)))\\
\quad\ snd = \varsigma(s,n)(Add.set2(n); Display.snd(n))], service)
\end{array}
$$

Fig. 5.9. Example: Fibonacci numbers

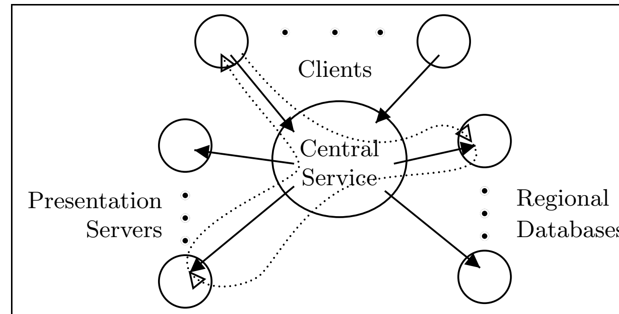


Fig. 5.10. Topology of a bank application

$$
\begin{array}{l}
let\ CentralService\ =\ [\ldots;\\
\quad regionalDatabase = \varsigma(s, ID)\ldots,\\
\quad presentationServer = \varsigma(s, device)\ \ldots,\\
\quad act = \varsigma(s, _)Repeat(Serve(getStatement)),\\
\quad getStatement = \varsigma(self, ID, device)\\
\quad\quad let\ state = (self.regionalDatabase(ID))\\
\quad\quad\quad\quad\quad\quad .getStatement(ID)\\
\quad\quad in\quad (self.presentationServer(device))\\
\quad\quad\quad\quad\quad\quad .getPresentation(state)\ \ ]
\end{array}
$$

Fig. 5.11. Example: bank account server

Part III

Semantics and Properties

6

Parallel Semantics

6.1 Structure of Parallel Activities

Each activity $\alpha[a;\sigma;\iota;F;R;f]$ is characterized by:

- a *current term* $(a = b \Uparrow f_i^{\gamma\to\alpha}, b')$ to be reduced: a contains the terms corresponding to the different requests being treated separated by $\Uparrow$. The left part b is the term currently evaluated, the right one $(f_i^{\gamma\to\alpha}, b')$ is the continuation: the future and term corresponding to a request that has been stopped before the end of its execution (because of a *Serve* primitive evaluated inside another service). Of course, b' can also contain continuations.
- a *store* (σ) that contains all the objects of the activity α. It can be considered as the memory associated to the activity α.
- an *active object location*: ι is the location of the active object of activity α; thus $\sigma(\iota)$ is the active object of activity α.
- *future values*: $F = \{f_i^{\gamma\to\alpha} \mapsto \iota\}$ is a list associating, for each served request, the corresponding future $f_i^{\gamma\to\alpha}$ and the location ι where the result of the request (also called future value) is stored.
- *pending requests*: $R = \{[m_j;\iota;f_i^{\gamma\to\alpha}]\}$, a list of pending requests. A request can be seen as the "reification" of a method call [150].

  Each request $r ::= [m_j;\iota;f_i^{\alpha\to\beta}]$ consists of:
  - the name of the *target method* m_j (invoked method),
  - the location of the *argument* passed to the request ι,
  - the *future* identifier which will be associated to the result $f_i^{\alpha\to\beta}$.
- a *current future*: $f = f_i^{\gamma\to\alpha}$, the future associated with the request currently served. To simplify notations, f will denote any future $(f ::= f_i^{\gamma\to\alpha})$.

$$P, Q ::= \alpha[a;\sigma;\iota;F;R;f] \parallel \beta[a';\sigma';\iota';F';R';f'] \parallel \ldots$$

$F :: \{f_i \mapsto \iota\}$ adds a new future association to the future values.

$$
\begin{array}{lll}
o ::= & [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} & \text{reduced object} \\
& |AO(\alpha) & \text{active object reference} \\
& |fut(f_i^{\alpha\to\beta}) & \text{future reference}
\end{array}
$$

6.2 Parallel Reduction

$$
\begin{aligned}
Repeat(a) &\triangleq [repeat = \varsigma(x)a; x.repeat()].repeat() \\
FifoService &\triangleq Repeat(Serve(\mathcal{M}))
\end{aligned}
$$

where $\mathcal{M}$ is the set of all method labels. Note that $\mathcal{M}$ only needs to contain all the method labels of the concerned (active) object.

$$\mathcal{R} ::= \ldots \mid Active(\mathcal{R}, m_j) \mid \mathcal{R} \Uparrow f, a$$

$$Repeat\ a\ Until\ b \triangleq [repeat = \varsigma(x)a; if\ (not(b))\ then\ x.repeat()].repeat()$$

6.2.1 More Operations on Store

- deep copy: $copy(\iota, \sigma)$,
- merge: $Merge(\iota, \sigma, \sigma')$,
- copy and merge: $Copy\&Merge(\sigma, \iota\ ;\ \sigma', \iota')$.

$\iota \in dom(copy(\iota, \sigma))$
$\iota' \in dom(copy(\iota, \sigma)) \Rightarrow locs(\sigma(\iota')) \subseteq dom(copy(\iota, \sigma))$
$\iota' \in dom(copy(\iota, \sigma)) \Rightarrow copy(\iota, \sigma)(\iota') = \sigma(\iota')$

Table 6.1. Deep copy

$$\vdash (\iota, \sigma)\ \text{OK} \Rightarrow \vdash (\iota, copy(\iota, \sigma))\ \text{OK}$$

$$
\begin{aligned}
&Merge(\iota, \sigma, \sigma') = \sigma'\theta + \sigma \\
&\quad \text{where } \theta = \{\!\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \backslash \{\iota\}, \iota'' \text{ fresh}\}\!\}
\end{aligned}
$$

Copy and merge

Definition 6.1 (Copy and merge)

$$Copy\&Merge(\sigma, \iota\ ;\ \sigma', \iota') \triangleq Merge(\iota', \sigma', copy(\iota, \sigma)\{\!\{\iota \leftarrow \iota'\}\!\})$$

Property 6.2 (Copy and merge)

$$\iota \in dom(\sigma') \wedge \iota \neq \iota' \Rightarrow \sigma'(\iota) = Copy\&Merge(\sigma, \iota''\ ;\ \sigma', \iota')(\iota)$$

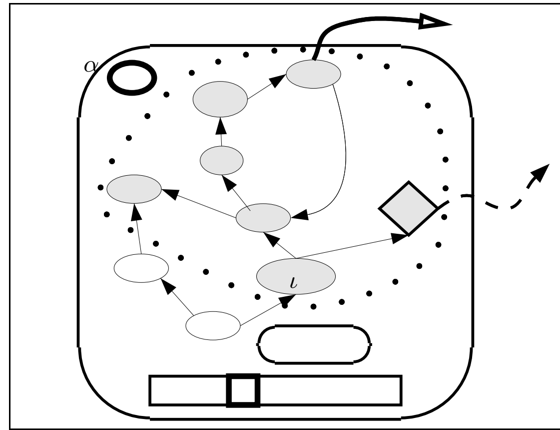


Fig. 6.1. Example of a deep copy: $copy(\iota, \sigma_\alpha)$

6.2.2 Reduction Rules

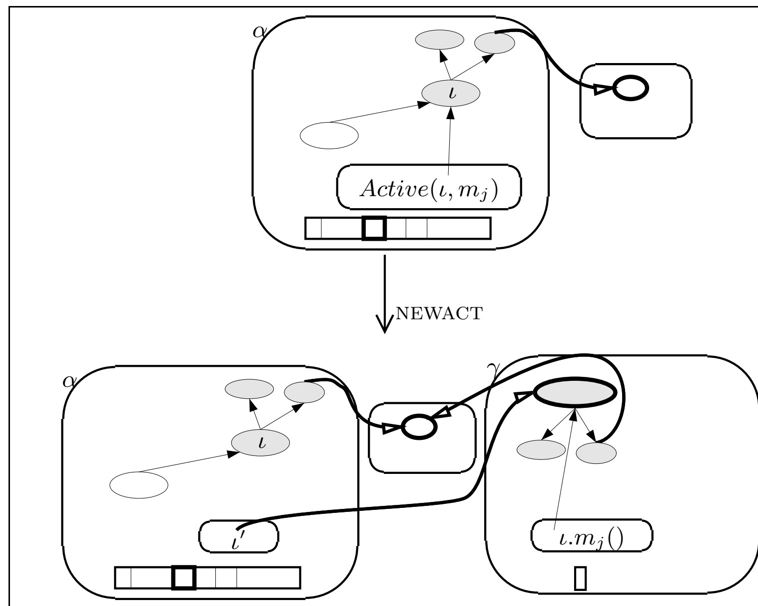


Fig. 6.2. NEWACT rule

LOCAL:

$$\frac{(a,\sigma)\rightarrow_S (a',\sigma') \quad \rightarrow_S \text{ does not clone a future}}{\alpha[a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f] \parallel P}$$

NEWACT:

$$\frac{\gamma \text{ fresh activity} \quad \iota' \notin dom(\sigma) \quad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \qquad \sigma_\gamma = copy(\iota'',\sigma) \quad Service = (\text{if } m_j = \emptyset \text{ then } FifoService \text{ else } \iota''.m_j())}{\alpha[\mathcal{R}[Active(\iota'',m_j)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[\mathcal{R}[\iota'];\sigma';\iota;F;R;f] \parallel \gamma[Service;\sigma_\gamma;\iota'';\emptyset;\emptyset;\emptyset] \parallel P}$$

REQUEST:

$$\frac{\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \quad f_i^{\alpha\to\beta} \text{ new future} \quad \iota_f \notin dom(\sigma_\alpha) \qquad \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\iota' \;;\; \sigma_\beta,\iota'') \quad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\to\beta})\} :: \sigma_\alpha}{\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta :: [m_j;\iota'';f_i^{\alpha\to\beta}];f_\beta] \parallel P}$$

SERVE:

$$\frac{R = R' :: [m_j;\iota_r;f'] :: R'' \quad m_j \in M \quad \forall m \in M,\, m \notin R'}{\alpha[\mathcal{R}[Serve(M)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[\iota.m_j(\iota_r) \Uparrow f, \mathcal{R}[[]];\sigma;\iota;F;R' :: R'';f'] \parallel P}$$

ENDSERVICE:

$$\frac{\iota' \notin dom(\sigma) \quad F' = F :: \{f \mapsto \iota'\} \quad \sigma' = Copy\&Merge(\sigma,\iota \;;\; \sigma,\iota')}{\alpha[\iota \Uparrow (f',a);\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a;\sigma';\iota;F';R;f'] \parallel P}$$

REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \quad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta,\iota_f \;;\; \sigma_\alpha,\iota)}{\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P}$$

Table 6.2. Parallel reduction (used or modified values are non-gray)

$$\begin{array}{lll} Active(Active(\iota,m_j),\emptyset) \longrightarrow & Active(\iota',\emptyset) & \longrightarrow \iota'' \\ & \sigma_\alpha(\iota') = AO(\beta) & \sigma_\gamma(\iota') = \sigma_\alpha(\iota') = AO(\beta) \\ & & \sigma_\alpha(\iota'') = AO(\gamma) \end{array}$$

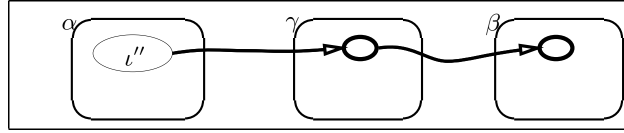


Fig. 6.3. A simple forwarder

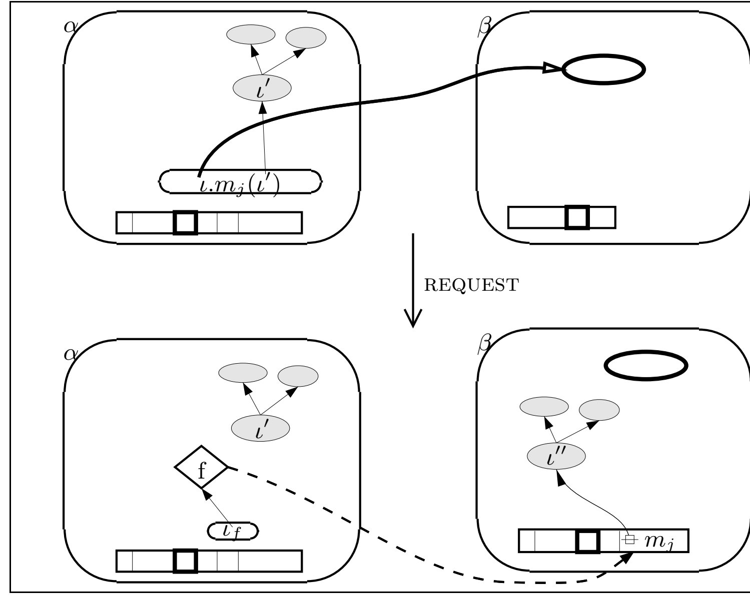


Fig. 6.4. REQUEST rule

REQUEST $\alpha = \beta$:

$$\frac{\sigma_\alpha(\iota) = AO(\alpha) \quad \iota'' \notin dom(\sigma_\alpha) \quad f_i^{\alpha\to\alpha} \text{ new future} \quad \iota_f \notin dom(\sigma_\alpha) \\ \iota'' \neq \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \{\iota_f \mapsto fut(f_i^{\alpha\to\alpha})\} :: \sigma_\alpha, \iota'')}{\alpha[\mathcal{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota_\alpha; F; R; f]\|\|Q \longrightarrow \alpha[\mathcal{R}[\iota_f]; \sigma'_\alpha; \iota_\alpha; F; R :: [m_j; \iota''; f_i^{\alpha\to\alpha}]; f]\|Q}$$

REPLY $\alpha = \beta$:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\alpha}) \quad F_\alpha(f_i^{\gamma\to\alpha}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\alpha, \iota_f \ ; \ \sigma_\alpha, \iota)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P}$$

6.3 Well-formedness

$$ActiveRefs(\alpha) = \{\beta | \exists \iota \in dom(\sigma_\alpha), \ \sigma_\alpha(\iota) = AO(\beta)\}$$
$$FutureRefs(\alpha) = \{f_i^{\beta\to\gamma} | \exists \iota \in dom(\sigma_\alpha), \ \sigma_\alpha(\iota) = fut(f_i^{\beta\to\gamma})\}$$

$$ActiveRefs(\alpha) = \{\beta, \delta\}$$
$$FutureRefs(\alpha) = \{f^{\alpha\to\beta}, f^{\beta\to\gamma}\}$$

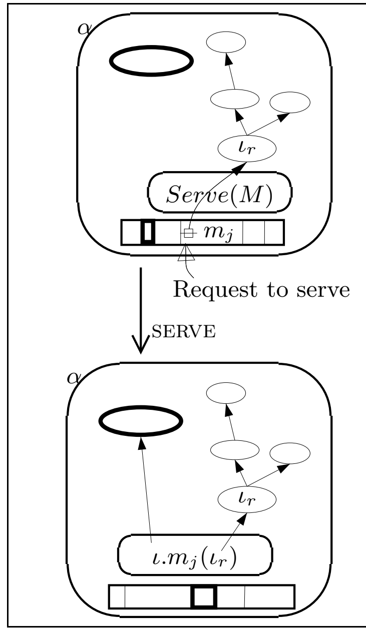


Fig. 6.5. SERVE rule

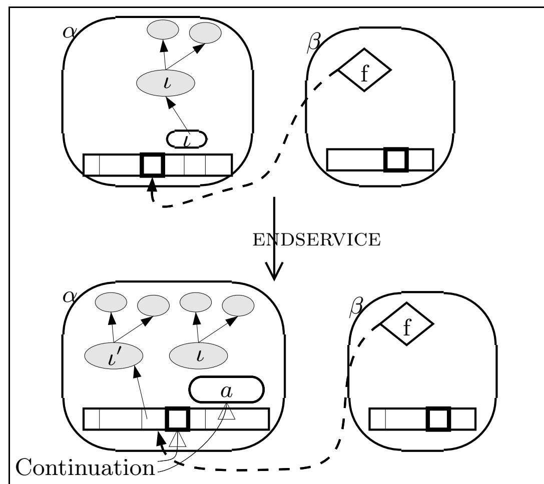


Fig. 6.6. ENDSERVICE rule

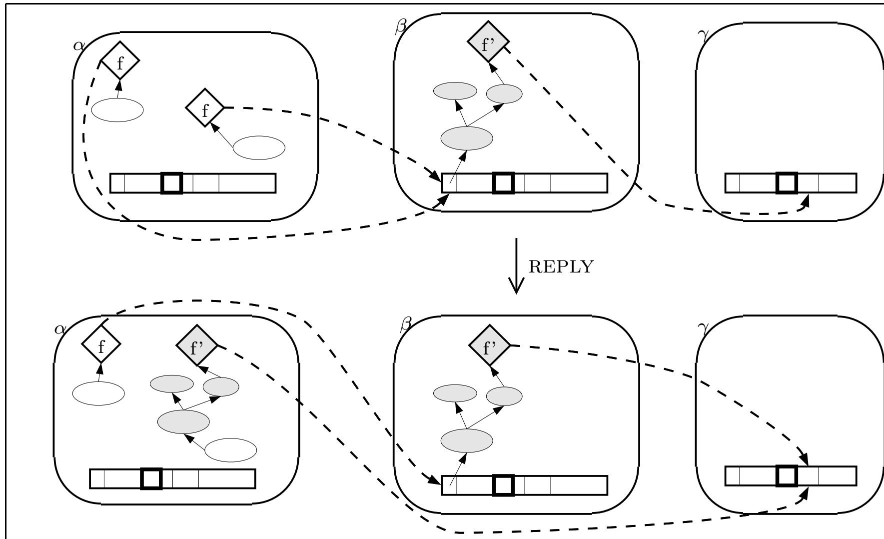


Fig. 6.7. REPLY rule

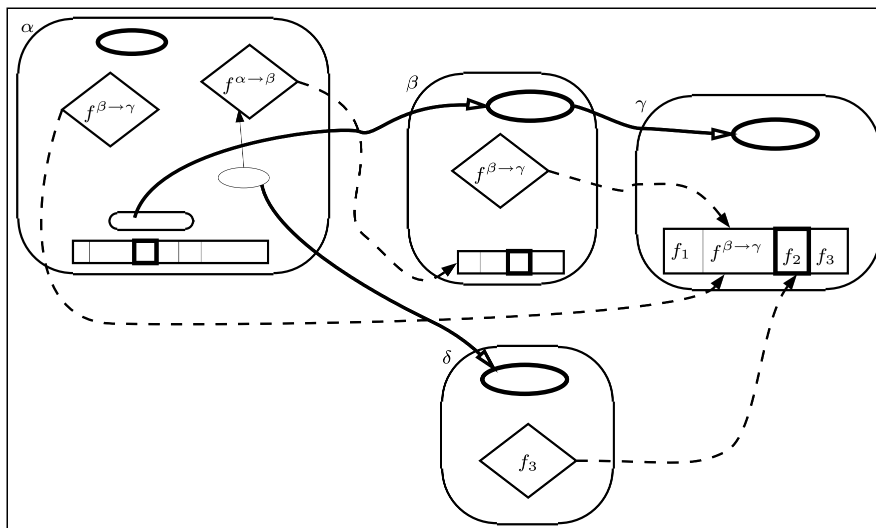


Fig. 6.8. Another example of configuration

Definition 6.3 (Future list) *Let $FL(\gamma)$ be:*

- *the list of futures that have been calculated,*
- *the current futures (the one in the activity and all those in the continuation of the current expression), and*
- *futures corresponding to pending requests of activity γ.*

Then:

$$\begin{aligned} FL(\gamma) = \{f_i^{\beta\to\gamma} | \{f_i^{\beta\to\gamma} \mapsto \iota\} \in F_\gamma\} & \\ :: \{f_\gamma\} :: \mathcal{F}(a_\gamma) & \\ :: \{f_i^{\beta\to\gamma} | [m_j, \iota, f_i^{\beta\to\gamma}] \in R_\gamma\} & \end{aligned}$$

$$\textit{where} \begin{cases} \mathcal{F}(a \Uparrow f, b) = f :: \mathcal{F}(b) \\ \mathcal{F}(a) = \emptyset \qquad\qquad \textit{if } a \neq a' \Uparrow f, b \end{cases}$$

$$FL(\gamma) = \{f_1, f^{\beta\to\gamma}, f_2, f_3\}$$

Definition 6.4 (Well-formedness)

$$\vdash P \text{ OK} \Leftrightarrow \forall \alpha \in P \begin{cases} \vdash (a_\alpha, \sigma_\alpha) \text{ OK} \\ \vdash (\iota_\alpha, \sigma_\alpha) \text{ OK} \\ \beta \in ActiveRefs(\alpha) \Rightarrow \beta \in P \\ f_i^{\beta\to\gamma} \in FutureRefs(\alpha) \Rightarrow f_i^{\beta\to\gamma} \in FL(\gamma) \end{cases}$$

Property 6.5 (Well-formed parallel reduction)

$$\vdash P \text{ OK} \wedge P \longrightarrow P' \quad \Longrightarrow \quad \vdash P' \text{ OK}$$

7

Basic ASP Properties

impς-calculus
ASP
Absence of sharing
Well-formed sequential reduction
Store partitioning
RSL compatibility
Well-formed parallel reduction
Equivalence modulo future updates
Local determinism
Configuration compatibility
Equivalence modulo future updates and reduction
Confluence
Deterministic Object Networks (DON)
. . .
Static DON (SDON)
Tree determinism (TDON)
. . .

Fig. 7.1. An informal property diagram

7.1 Notation and Hypothesis

Potential Services

$$P_0 \xrightarrow{*} P \ \wedge\ a_{\alpha_P} = \mathcal{R}[Serve(M)] \Rightarrow M \in M_0$$

Definition 7.1 (Potential services)
Let P_0 be an initial configuration. $\mathcal{M}_{\alpha_P}$ is any set verifying:

$$P \xrightarrow{*} P' \ \wedge\ a_{\alpha_{P'}} = \mathcal{R}[Serve(M)] \Rightarrow \exists M' \in \mathcal{M}_{\alpha_P},\ M \subseteq M'$$

7.2 Object Sharing

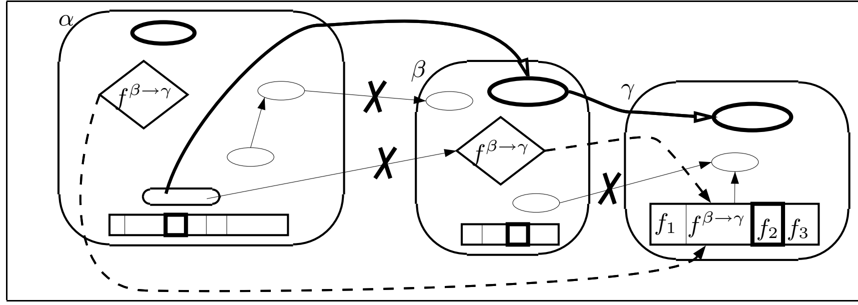


Fig. 7.2. Absence of sharing

7.3 Isolation of Futures and Parameters

Property 7.2 (Store partitioning)
Let:

$$ActiveStore(\alpha) = copy(\iota_\alpha, \sigma_\alpha) \cup \bigcup_{\iota \in locs(a_\alpha)} copy(\iota, \sigma_\alpha)$$

At any stage of computation, each activity has the following invariant:

$$\sigma_\alpha \supseteq \left(ActiveStore(\alpha) \bigoplus_{\{f \mapsto \iota_f\} \in F_\alpha} copy(\iota_f, \sigma_\alpha) \bigoplus_{[m_j;\iota_r;f] \in R_\alpha} copy(\iota_r, \sigma_\alpha) \right)$$

where $\bigoplus$ is the disjoint union.

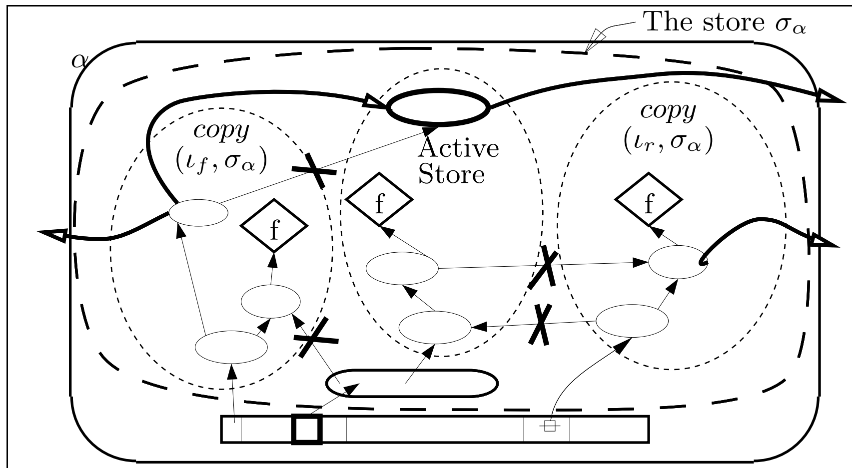


Fig. 7.3. Store partitioning: future value, active store, request parameter

8

Confluence Property

8.1 Configuration Compatibility

Definition 8.1 (Request Sender List) *The ith element of $RSL(\alpha)$ is defined by:*

$$(RSL(\alpha))_i = \beta^f \text{ if } f_i^{\beta\to\alpha} \in FL(\alpha)$$

$$RSL(\delta) = \beta^{bar} :: \alpha^{foo} :: \gamma^{gee} :: \gamma^{gee} :: \alpha^{foo} :: \beta^{foo} :: \beta^{bar} :: \gamma^{gee}$$

$$RSL(\delta)\big|_{foo,bar} = \beta^{bar} :: \alpha^{foo} :: \alpha^{foo} :: \beta^{foo} :: \beta^{bar}$$

Definition 8.2 (RSL comparison $\trianglelefteq$)
RSLs are ordered by the prefix order on activities:

$$\alpha_1{}^{f_1} \dots \alpha_n{}^{f_n} \trianglelefteq \alpha_1'{}^{f_1'} \dots \alpha_m'{}^{f_m'} \Leftrightarrow \begin{cases} n \leq m \\ \forall i \in [1..n], \alpha_i = \alpha_i' \end{cases}$$

Definition 8.3 (RSL compatibility: $RSL(\alpha) \bowtie RSL(\beta)$)
Two RSLs are compatible if they have a least upper bound or equivalently if one is a prefix of the other:

$$\begin{aligned} RSL(\alpha) \bowtie RSL(\beta) \Leftrightarrow &\quad RSL(\alpha) \sqcup RSL(\beta) \text{ exists} \\ \Leftrightarrow &\quad (RSL(\alpha) \trianglerighteq RSL(\beta)) \vee (RSL(\beta) \trianglelefteq RSL(\alpha)) \end{aligned}$$

Definition 8.4 (Configuration compatibility: $P \bowtie Q$)
If P_0 is a configuration such that $P_0 \xrightarrow{*} P$ and $P_0 \xrightarrow{*} Q$:

$$P \bowtie Q \Leftrightarrow \forall \alpha \in P \cap Q, \forall M \in \mathcal{M}_{\alpha_{P_0}}, RSL(\alpha_P)\big|_M \bowtie RSL(\alpha_Q)\big|_M$$

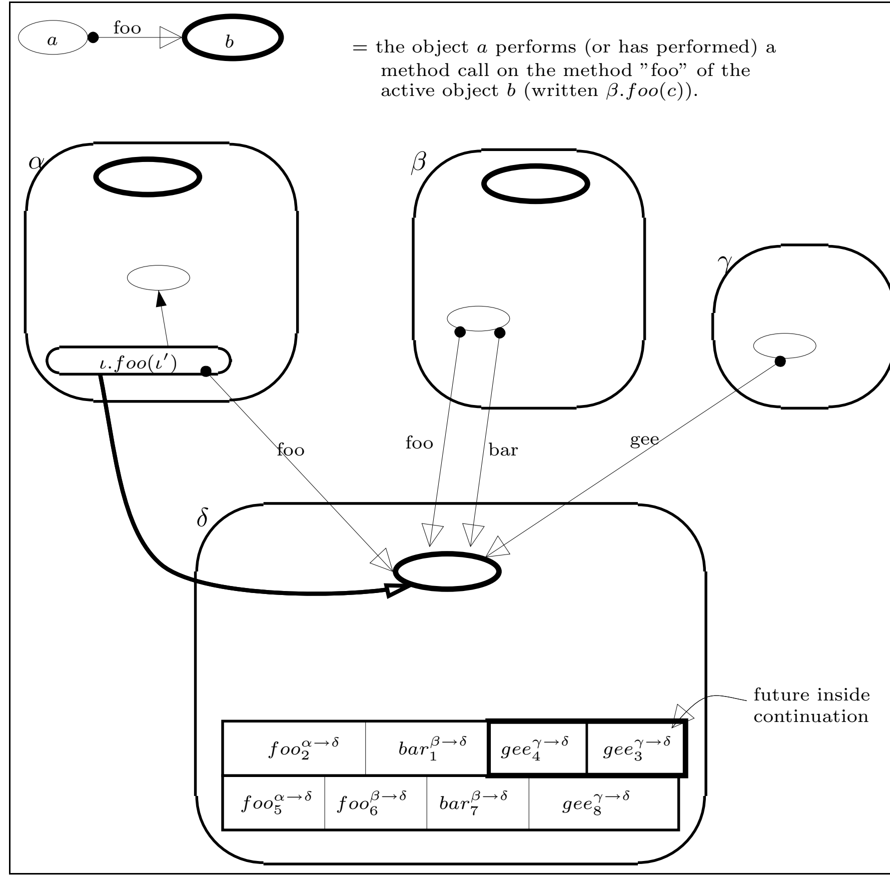


Fig. 8.1. Example of RSL

$$\mathcal{M}_{\delta_{P_0}} = \{\{foo, bar\}, \{foo, gee\}\}$$
$$RSL_3(\delta) = \ldots \alpha^{foo} :: \gamma^{gee} :: \beta^{foo} :: \gamma^{gee} :: \beta^{bar}$$
$$RSL_4(\delta) = \ldots \alpha^{foo} :: \beta^{foo} :: \gamma^{gee} :: \beta^{bar} :: \beta^{bar}$$

$$RSL_3(\delta)\big|_{bar,foo} = \ldots \alpha :: \beta :: \beta \bowtie \ldots \alpha :: \beta :: \beta :: \beta = RSL_4(\delta)\big|_{bar,foo}$$

$$RSL_3(\delta)\big|_{gee,foo} = \ldots \alpha :: \beta :: \gamma :: \gamma \bowtie \ldots \alpha :: \beta :: \gamma = RSL_4(\delta)\big|_{gee,foo}$$

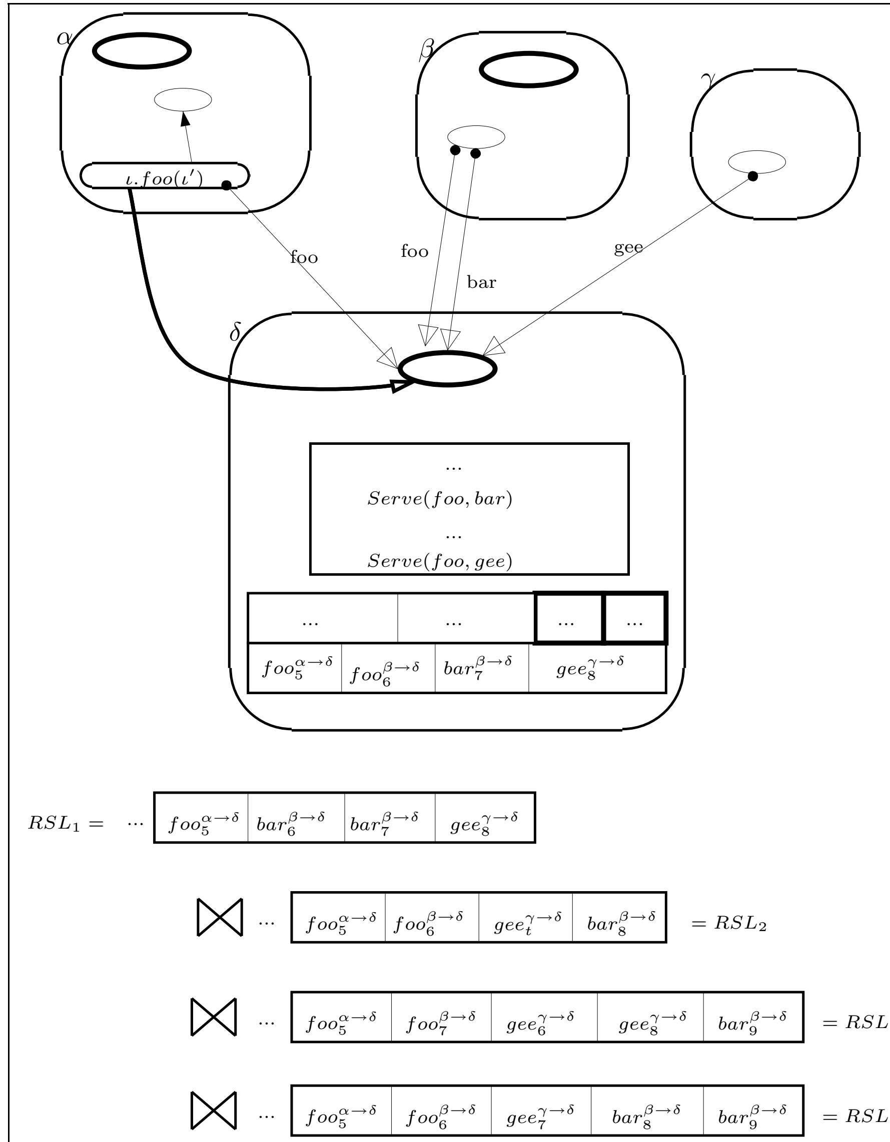


Fig. 8.2. Example of RSL compatibility

8.2 Equivalence Modulo Future Updates

8.2.1 Principles

8.2.2 Alias Condition

Aliases existing in P_1 *also exist in* P_2 if P_1 and P_2 verify the following relation:

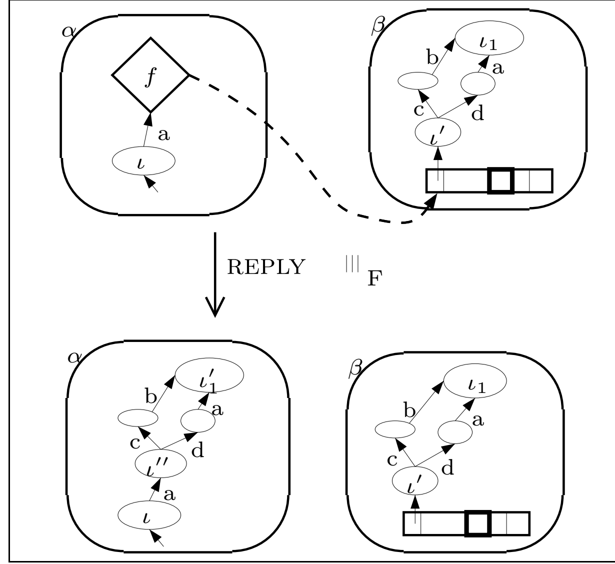


Fig. 8.3. Two equivalent configurations modulo future updates

$$\exists L_1, L_2 \begin{cases} \alpha \overset{\alpha}{\mapsto}_{L_1} \iota'_1 \\ \alpha \overset{\alpha}{\mapsto}_{L_2} \iota'_1 \end{cases} \text{ in } P_1 \;\Rightarrow \exists L, L', L'_1, L'_2, \iota' \begin{cases} L_1 = L.L'_1 \\ L_2 = L'.L'_2 \\ L'_1 \neq \emptyset \\ \alpha \overset{\alpha *}{\mapsto}_{L} \iota' \overset{\beta}{\mapsto}_{L'_1} \iota_1 \\ \alpha \overset{\alpha *}{\mapsto}_{L'} \iota' \overset{\beta}{\mapsto}_{L'_2} \iota_1 \end{cases} \text{ in } P_2$$

$$\exists L_1, L_2 \; \alpha \overset{\alpha}{\mapsto}_{\ldots} \iota \overset{\alpha}{\mapsto}_{L_1} \iota'_1 \wedge \alpha \overset{\alpha}{\mapsto}_{\ldots} \iota \overset{\alpha}{\mapsto}_{L_2} \iota'_1$$

$$\exists L, L', L'_1, L'_2, \iota' \begin{cases} L_1 = L.L'_1 \wedge L_2 = L'.L'_2 \wedge L'_1 \neq \emptyset \\ \alpha \overset{\alpha}{\mapsto}_{\ldots} \iota \overset{\alpha *}{\mapsto}_{L} \iota' \overset{\beta}{\mapsto}_{L'_1} \iota_1 \\ \alpha \overset{\alpha}{\mapsto}_{\ldots} \iota \overset{\alpha *}{\mapsto}_{L'} \iota' \overset{\beta}{\mapsto}_{L'_2} \iota_1 \end{cases}$$

$$\iota' \overset{\beta}{\mapsto}_{L'_1} \iota_1 \wedge \iota' \overset{\beta}{\mapsto}_{L'_2} \iota_1$$

8.2.3 Sufficient Conditions

$$P \xrightarrow{\text{REPLY}} P' \Rightarrow P \equiv_F P'$$

$$\left(P_1 \xrightarrow{\text{REPLY}} P' \wedge P_2 \xrightarrow{\text{REPLY}} P'\right) \quad \Rightarrow \quad P_1 \equiv_F P_2$$

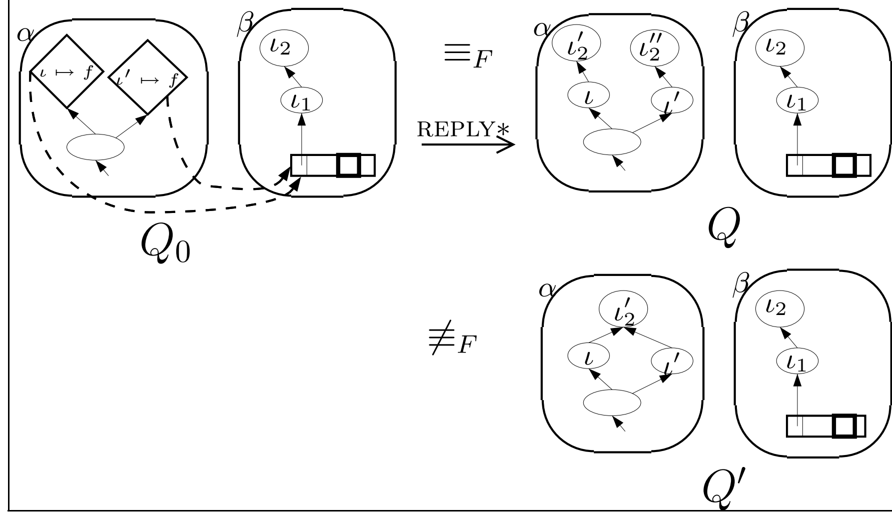


Fig. 8.4. An example illustrating the alias condition

Definition 8.5 (Cycle of futures)
A configuration contains a cycle of futures if some futures values are mutually dependent: each future value references another future, finally leading to a cycle.

A cycle of futures is a set of future identifiers $\{fut(f_i^{\gamma_i \to \beta_n})\}$ *such that* $\beta_0 \ldots \beta_n$ *verify:*

$$\begin{cases} \left(\forall i,\, 0 < i \leq n, fut(f_{i-1}^{\gamma_{i-1} \to \beta_{i-1}}) \in copy(F_{\beta_i}(fut(f_i^{\gamma_i \to \beta_i})), \sigma_{\beta_i})\right) \\ fut(f_n^{\gamma_n \to \beta_n}) \in copy(F_{\beta_0}(fut(f_0^{\gamma_0 \to \beta_0})), \sigma_{\beta_0}) \end{cases}$$

$$P_1 \xrightarrow{\text{REPLY}} P' \wedge P_2 \xrightarrow{\text{REPLY}} P'$$

8.3 Properties of Equivalence Modulo Future Updates

$$T \in \{\text{LOCAL}, \text{NEWACT}, \text{REQUEST}, \text{SERVE}, \text{ENDSERVICE}, \text{REPLY}\}$$

Definition 8.6 (Parallel reduction modulo future updates)

$$\overset{T}{\Longrightarrow} \quad = \quad \begin{array}{ll} \xrightarrow{\text{REPLY}*}\xrightarrow{T} & \textit{if } T \neq \text{REPLY} \\ \xrightarrow{\text{REPLY}*} & \textit{if } T = \text{REPLY} \end{array}$$

Property 8.7 (Equivalence modulo future updates and reduction)

$$P \xrightarrow{T} Q \;\wedge\; P \equiv_F P' \quad \Rightarrow \quad \exists Q',\, P' \overset{T}{\Longrightarrow} Q' \;\wedge\; Q' \equiv_F Q$$

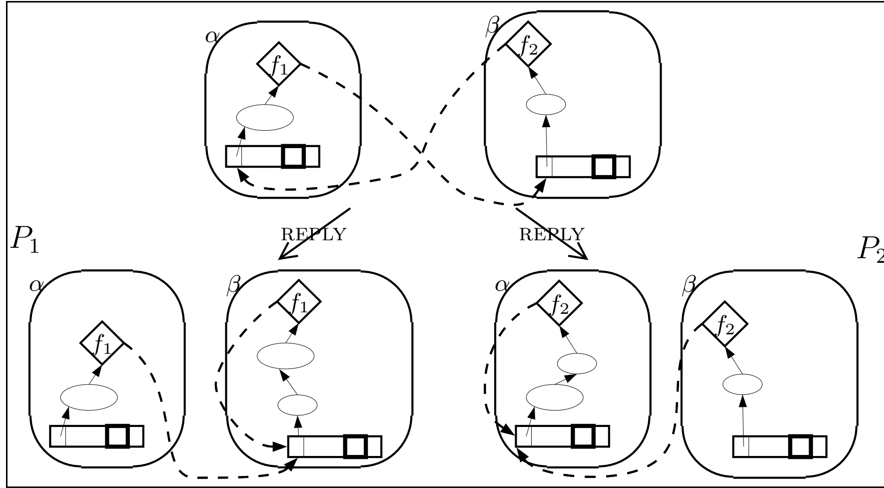


Fig. 8.5. Updates in a cycle of futures

Property 8.8 (Equivalence and generalized parallel reduction)

$$P \overset{T}{\Longrightarrow} Q \;\wedge\; P \equiv_F P' \quad\Rightarrow\quad \exists Q',\, P' \overset{T}{\Longrightarrow} Q' \;\wedge\; Q' \equiv_F Q$$

8.4 Confluence

Definition 8.9 (Confluent configurations: $P_1 \curlyvee P_2$)

$$P_1 \curlyvee P_2 \Leftrightarrow \exists R_1, R_2, \begin{cases} P_1 \xrightarrow{*} R_1 \\ P_2 \xrightarrow{*} R_2 \\ R_1 \equiv_F R_2 \end{cases}$$

Theorem 8.10 (Confluence)

$$\begin{cases} P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \\ Q_1 \bowtie Q_2 \end{cases} \Longrightarrow Q_1 \curlyvee Q_2$$

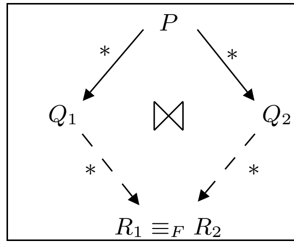


Fig. 8.6. Confluence

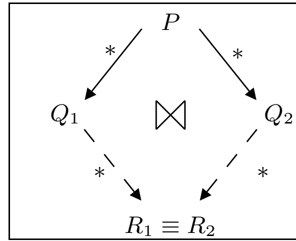


Fig. 8.7. Confluence without cycle of futures

9

Determinacy

9.1 Deterministic Object Networks

Definition 9.1 (DON)
A configuration P, derived from an initial configuration P_0, is a Deterministic Object Network, $DON(P)$, if it cannot be reduced to a configuration where two interfering requests can be sent concurrently to the same destination activity:

$$DON(P) \Leftrightarrow \left(\begin{array}{r} P \xrightarrow{*} Q \Rightarrow \forall \alpha \in Q,\ \forall M \in \mathcal{M}_{\alpha_{P_0}}, \exists^1 \beta \in Q, \exists m \in M, \\ a_\beta = \mathcal{R}[\iota.m(\ldots)] \wedge \sigma_\beta(\iota) = AO(\alpha) \end{array} \right)$$

where $\exists^1$ means "there is at most one."

Property 9.2 (DON and compatibility)

$$DON(P) \wedge P \xrightarrow{*} Q_1 \wedge P \xrightarrow{*} Q_2 \Rightarrow Q_1 \bowtie Q_2$$

$$a_\beta = \mathcal{R}[\iota.foo(\ldots)] \wedge \sigma_\beta(\iota) = AO(\alpha)$$
$$a_\gamma = \mathcal{R}[\iota'.bar(\ldots)] \wedge \sigma_\gamma(\iota') = AO(\alpha)$$

Theorem 9.3 (DON determinism)

$$\begin{cases} DON(P) \\ P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \end{cases} \Longrightarrow \ Q_1 \curlyvee Q_2$$

9.2 Toward a Static Approximation of DON Terms

If dynamically a request on the method foo can be sent from α to β, and $\dot\alpha$ and $\dot\beta$ are the static approximations of activities α and β then $(\dot\alpha, \dot\beta, foo) \in \mathcal{G}(P_0)$:

$$P \xrightarrow{*} Q \wedge a_{\alpha_Q} = \mathcal{R}[\iota.foo(\iota')] \wedge \sigma_{\alpha_Q}(\iota) = AO(\beta) \Rightarrow (\dot\alpha, \dot\beta, foo) \in \mathcal{G}(P)$$

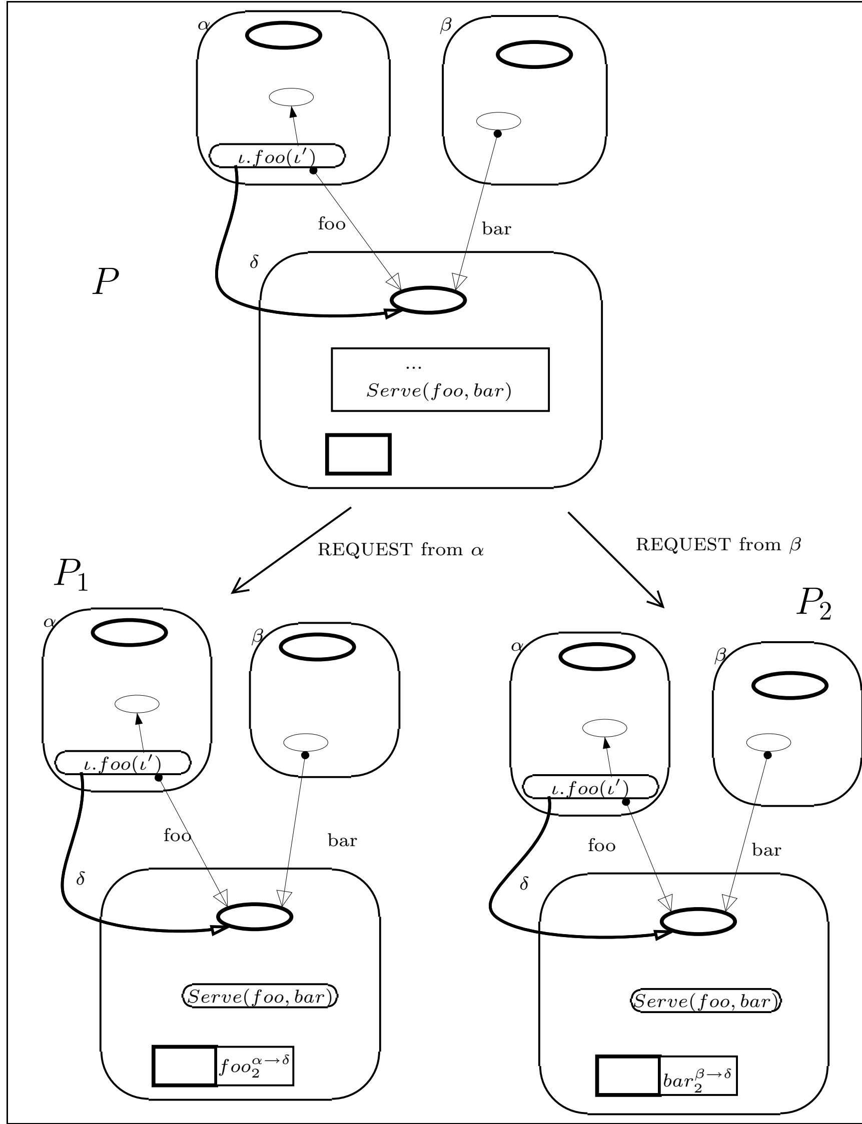


Fig. 9.1. A non-DON term

Definition 9.4 (Static DON) *A program P_0 is a Static Deterministic Object Network $SDON(P_0)$ if it verifies:*

$$SDON(P_0) \Leftrightarrow \left(\begin{cases} (\dot{\alpha}, \dot{\beta}, m_1) \in \mathcal{G}(P_0) \\ (\dot{\alpha}', \dot{\beta}, m_2) \in \mathcal{G}(P_0) \\ \dot{\alpha} \neq \dot{\alpha}' \end{cases} \Rightarrow \forall M \in \mathcal{M}_{\dot{\beta}_{P_0}},\ \{m_1, m_2\} \not\subseteq M \right)$$

where $\forall \beta,\ M \in \mathcal{M}_{\beta_P} \Rightarrow M \in \mathcal{M}_{\dot{\beta}_P}$.

Of course, Static DON terms are DON terms and behave deterministically:

Property 9.5 (Static approximation)

$$SDON(P_0) \Rightarrow DON(P_0)$$

Theorem 9.6 (SDON determinism)
SDON terms behave deterministically:

$$\begin{cases} SDON(P) \\ P \xrightarrow{*} Q_1 \\ P \xrightarrow{*} Q_2 \end{cases} \Longrightarrow \; Q_1 \curlyvee Q_2$$

9.3 Tree Topology Determinism

Theorem 9.7 (Tree determinacy, TDON)
If, at every step of the reduction, the request flow graph forms a set of trees then the reduction is deterministic (TDON).

9.4 Deterministic Examples

9.4.1 The Binary Tree

$$\begin{array}{l} let\ tree = (BT.new).add(3,4).add(2,3).add(5,6).add(7,8)\ in \\ \quad [a = tree.search(5), b = tree.search(3)].b := tree.search(7) \end{array}$$

$$\begin{array}{l} let\ tree = (BT.new).add(3,4).add(2,3).add(5,6).add(7,8)\ in \\ \quad let\ Client = [a = tree.search(5), b = tree.search(3)]\ in \\ \qquad Client.b := tree.search(7);\ tree.add(1, Client.a) \end{array}$$

9.4.2 The Fibonacci Number Example

Compatibility

$$\begin{array}{l} RSL(\gamma) = \emptyset \mid \beta^{set2} \mid \alpha^{set1} \mid \\ \qquad \alpha^{set1} :: \beta^{set2} :: RSL(\gamma) \mid \beta^{set2} :: \alpha^{set1} :: RSL(\gamma) \end{array}$$

Fibonacci Example Is a DON

SDON Approximation

9.5 Discussion: Comparing Request Service Strategies

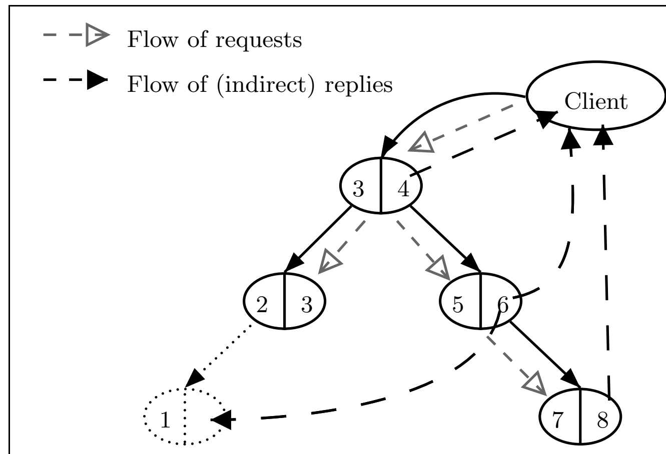


Fig. 9.2. Concurrent replies in the binary tree case

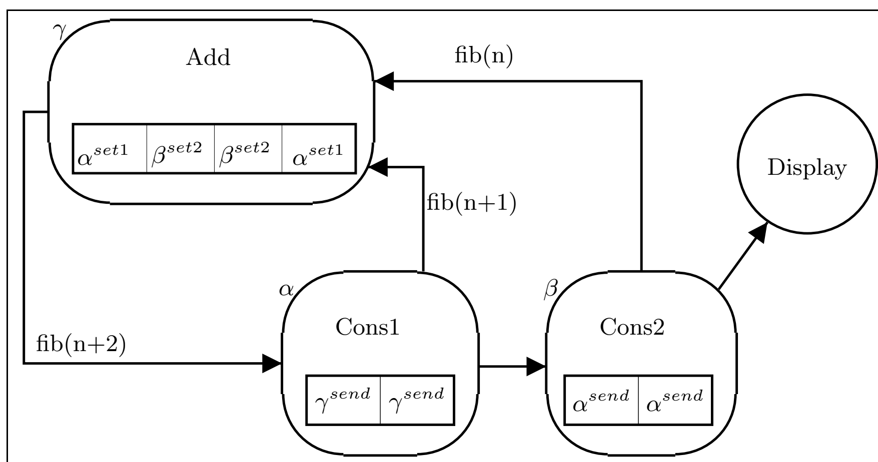


Fig. 9.3. Fibonacci number RSLs

Part IV

A Few More Features

10

More Confluent Features

10.1 Delegation

Principles of Explicit Delegation

Parallel DELEGATE:

$$\frac{\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \sigma_\beta, \iota'') \quad f_\emptyset \text{ new future}}{\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_i^{\gamma\rightarrow\alpha}] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[[]]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\emptyset] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\gamma\rightarrow\alpha}]; f_\beta] \parallel P}$$

Sequential DELEGATE:

$$\frac{\sigma_\alpha(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \quad k \in 1..m}{\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P}$$

Table 10.1. Rules for delegation (DELEGATE)

Parallel DELEGATE:

$$\frac{\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota' \ ; \ \sigma_\beta, \iota'') \quad \mathcal{R} \text{ does not contain any continuation}}{\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))] \Uparrow (f, a); \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_i^{\gamma\rightarrow\alpha}] \parallel \\ \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[a; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\gamma\rightarrow\alpha}]; f_\beta] \parallel P}$$

Generalized REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\delta}) \quad F_\beta(f_i^{\gamma\to\delta}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota)}{\begin{array}{c}\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P\end{array}}$$

Definition 10.1 (Well-formedness)

$$\vdash P\ \text{OK} \Leftrightarrow \forall \alpha \in P \begin{cases} \vdash (a_\alpha, \sigma_\alpha)\ \text{OK} \\ \vdash (\iota_\alpha, \sigma_\alpha)\ \text{OK} \\ \beta \in ActiveRefs(\alpha) \Rightarrow \beta \in P \\ f_i^{\beta\to\gamma} \in FutureRefs(\alpha) \Rightarrow \exists \delta,\ f_i^{\beta\to\gamma} \in FL(\delta) \end{cases}$$

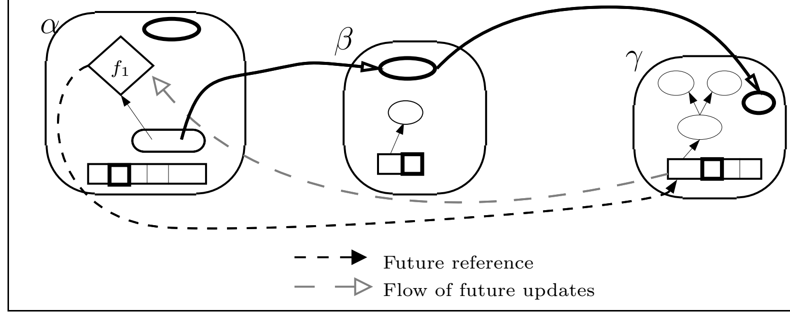


Fig. 10.1. Explicit delegation in ASP

Implicit Delegation in ASP

REQUEST:

$$\frac{\begin{array}{c}\sigma_\beta(\iota) = AO(\gamma) \quad \iota'' \notin dom(\sigma_\gamma) \quad f_i^{\beta\to\gamma}\ \text{new future} \quad \iota_f \notin dom(\sigma_\beta) \\ \sigma'_\gamma = Copy\&Merge(\sigma_\beta, \iota'\ ;\ \sigma_\gamma, \iota'') \quad \sigma'_\beta = \{\iota_f \mapsto fut(f_i^{\beta\to\gamma})\} :: \sigma_\beta\end{array}}{\begin{array}{l}\beta[\mathcal{R}[\iota.m_j(\iota')]; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_i^{\alpha\to\beta}] \parallel \gamma[a_\gamma; \sigma_\gamma; \iota_\gamma; F_\gamma; R_\gamma; f_\gamma] \parallel P \longrightarrow \\ \beta[\mathcal{R}[\iota_f]; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta; f_i^{\alpha\to\beta}] \parallel \gamma[a_\gamma; \sigma'_\gamma; \iota_\gamma; F_\gamma; R_\gamma :: [m_j; \iota''; f_i^{\beta\to\gamma}]; f_\gamma] \parallel P\end{array}}$$

ENDSERVICE:

$$\frac{\iota'_f \notin dom(\sigma_\beta) \quad F'_\beta = F_\beta :: \{f_i^{\alpha\to\beta} \mapsto \iota'_f\} \quad \sigma' = Copy\&Merge(\sigma'_\beta, \iota_f\ ;\ \sigma''_\beta, \iota'_f)}{\beta[\iota_f \Uparrow (f', a); \sigma'_\beta; \iota_\beta; F_\beta; R_\beta; f_i^{\alpha\to\beta}] \parallel P \longrightarrow \beta[a; \sigma''_\beta; \iota; F'_\beta; R; f'] \parallel P}$$

Then future $fut(f_i^{\beta\to\gamma})$ becomes an alias for future $f_i^{\delta\to\beta}$, and this alias information must be transmitted to all the potential users of the result by the REPLY rules, e.g., α:

REPLY:

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = fut(f_i^{\alpha\to\beta}) \quad F'_\beta(f_i^{\gamma\to\beta}) = \iota_f \\ \sigma''_\beta(\iota'_f) = fut(f_i^{\beta\to\gamma}) \quad \sigma'_\alpha = \{\iota \to fut(f_i^{\beta\to\gamma})\} + \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[a_\alpha;\boldsymbol{\sigma_\alpha};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\boldsymbol{\sigma''_\beta};\iota_\beta;\boldsymbol{F'_\beta};R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha;\boldsymbol{\sigma'_\alpha};\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\boldsymbol{\sigma''_\beta};\iota_\beta;\boldsymbol{F'_\beta};R_\beta;f_\beta] \parallel P\end{array}}$$

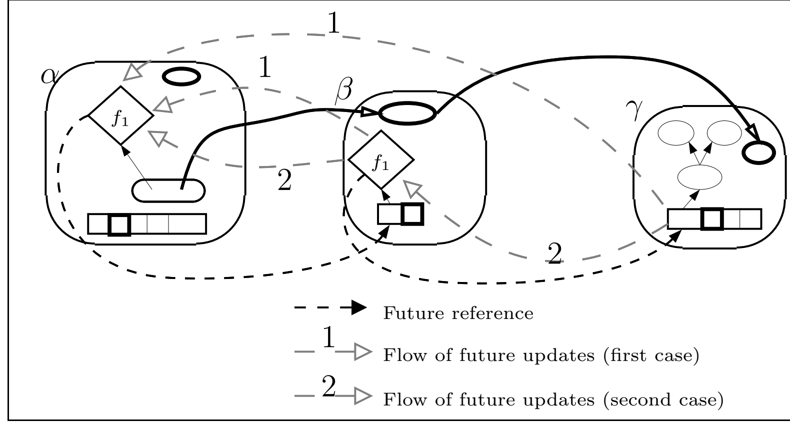


Fig. 10.2. Implicit delegation in ASP

10.2 Explicit Wait

$$[\![[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}]\!] \triangleq [wait = [], l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$$

$$[\![waitFor(a)]\!] \triangleq a.wait$$

10.3 Method Update

$$[foo_field = \lambda self.\lambda y.body, foo = \varsigma(x, y)\,((x.foo_field\ x)\ y)]$$

$$x.foo \Leftarrow b \triangleq x.foo_field := b$$

11

Non-Confluent Features

11.1 Testing Future Reception

WAITT:

$$\frac{\sigma(\iota) = fut(f_i^{\alpha\to\beta})}{(\mathcal{R}[awaited(\iota)], \sigma) \to_S (\mathcal{R}[true], \sigma)}$$

WAITF:

$$\frac{\sigma(\iota) \neq fut(f_i^{\alpha\to\beta})}{(\mathcal{R}[awaited(\iota)], \sigma) \to_S (\mathcal{R}[false], \sigma)}$$

A Simple Example: Synchronization on Available Futures

$let\ f_1 = alpha.fib(n)\ in$
$let\ f_2 = beta.fib(n)\ in$
$\quad Repeat$
$\quad\quad if\ not(awaited(f_1))\ then\ s.result := foo(f_1)$
$\quad\quad\quad else\ if\ not(awaited(f_2))\ then\ s.result := foo(f_2)$
$\quad Until\ \ (not(awaited(f_1)) \vee not(awaited(f_2)))$

11.2 Non-blocking Services

SERVEWBSERVE:

$$\frac{R = R' :: [m_j; \iota_r; f'] :: R'' \quad m_j \in M \quad \forall m \in M,\, m \notin R'}{\begin{array}{c}\alpha[\mathcal{R}[ServeWithoutBlocking(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \\ \alpha[\iota.m_j(\iota_r) \Uparrow f, \mathcal{R}[[]]; \sigma; \iota; F; R' :: R''; f'] \parallel P\end{array}}$$

ServeWBContinue:

$$\frac{\forall m \in M,\, m \notin R}{\alpha[\mathcal{R}[ServeWithoutBlocking(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[[]]; \sigma; \iota; F; R; f] \parallel P}$$

11.3 Testing Request Reception

inQueueT:

$$\frac{\exists m \in M,\, m \in R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[true]; \sigma; \iota; F; R; f] \parallel P}$$

inQueueF:

$$\frac{\forall m \in M,\, m \notin R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[false]; \sigma; \iota; F; R; f] \parallel P}$$

$$ServeWithoutBlocking(M) \triangleq if\ inQueue(M)\ then\ Serve(M)\ else\ []$$

11.4 Join Patterns

11.4.1 Translating Join Calculus Programs

$$\begin{array}{l} Cell \triangleq Active\big([s_v = [],\, set_v = []; \\ \qquad set = \varsigma(this, v) this.set_v := v \\ \qquad s = \varsigma(this, v) this.s_v := v \\ \qquad get = \varsigma(this)[] \\ \qquad srv = \varsigma(this) Repeat(if\ inQueue(s) \wedge inQueue(set)\ then \\ \qquad\qquad\qquad this.setcell() \\ \qquad\qquad if\ inQueue(s) \wedge inQueue(get)\ then \\ \qquad\qquad\qquad this.getcell()), \\ \qquad setcell() = \varsigma(this)(Serve(set); Serve(s); thisActivity.s(this.set_v)), \\ \qquad getcell() = \varsigma(this)(Serve(get); Serve(s); thisActivity.s(this.s_v); this.s_v)\big) \end{array}$$

Fig. 11.1. A join-calculus cell in ASP

$$Cell.s([]); Cell.set([x = 2]); Cell.get()$$

11.4.2 Extended Join Services in ASP

$$
\begin{aligned}
Join((m_1, m_2), (m_1, m_3)) \triangleq\ & let\ served = false\ in \\
& Repeat \\
& \quad if\ (inQueue(m_1) \wedge inQueue(m_2))\ then \\
& \qquad (Serve(m_1);\ Serve(m_2);\ served := true) \\
& \quad else\ if\ (inQueue(m_1) \wedge inQueue(m_3))\ then \\
& \qquad (Serve(m_1);\ Serve(m_3);\ served := true) \\
& Until(served = true)
\end{aligned}
$$

$$
\begin{aligned}
& Join((m_{11}, m_{12}, \ldots, m_{1n_1}), (m_{21}, \ldots, m_{2n_2}), \ldots (m_{k1}, \ldots, m_{kn_k})) \triangleq \\
& \qquad let\ served = false\ in \\
& \qquad Repeat \\
& \qquad\quad if\ (inQueue(m_{11}) \wedge inQueue(m_{12}) \wedge \ldots \wedge inQueue(m_{1n_1}))\ then \\
& \qquad\qquad (Serve(m_{11});\ Serve(m_{12}); \ldots Serve(m_{1n_1});\ served := true) \\
& \qquad\quad else\ if\ (inQueue(m_{21}) \wedge \ldots \wedge inQueue(m_{2n_2}))\ then \\
& \qquad\qquad (Serve(m_{21}); \ldots Serve(m_{2n_2});\ served := true) \\
& \qquad\quad else\ if\ (inQueue(m_{k1}) \wedge \ldots \wedge inQueue(m_{kn_k}))\ then \\
& \qquad\quad \ldots \\
& \qquad\qquad (Serve(m_{k1}); \ldots Serve(m_{kn_k});\ served := true) \\
& \qquad Until(served = true)
\end{aligned}
$$

12

Migration

12.1 Migrating Active Objects

$$Migrate = \varsigma(this)\, let\ newao = Active(this, sevice)\ in \\ (CreateForwarders(newao); FifoService)$$

where *service* is the service method to be executed when the activity has migrated, and $CreateForwarders(newao)$ replaces the body of each method of the current active object by a forwarded call:

$$CreateForwarders(newao) \triangleq \forall m_j,\ m_j \Leftarrow \varsigma(x,y) newao.m_j(y)$$

$$surrogate = \varsigma(s) s.alias\langle s.clone\rangle$$

12.2 Optimizing Future Updates

$$CreateForwarders(newao) \triangleq \forall m_j,\ m_j \Leftarrow \varsigma(x,y)\, delegate(newao.m_j(y))$$

12.3 Migration and Confluence

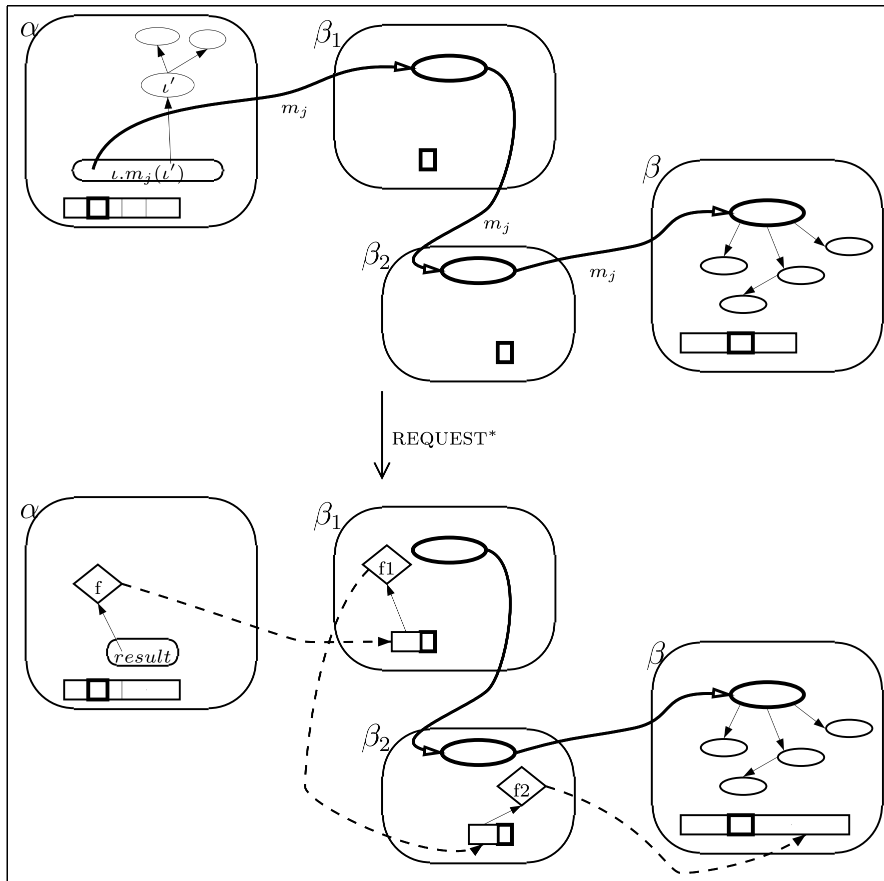


Fig. 12.1. Chain of method calls and chain of corresponding futures

13

Groups

13.1 Groups in an Object Calculus

Group Constructor

$a, b \in L ::= \ldots,$	ASP without groups
$\mid Group(a_k)^{k \in 1..l}$	group constructor

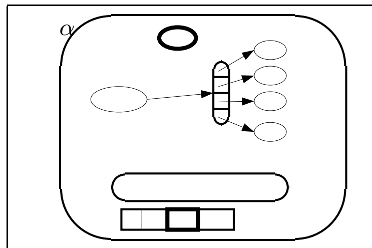


Fig. 13.1. A group of passive objects

Groups as an Extension of Local Semantics: A Group Proxy

$o ::= \ldots$	reduced object, or active object or future reference
$\mid Gr(\iota_k)^{k \in 1..l}$	group of references

$$\mathcal{R} ::= \ldots \mid Group(\iota_k, \mathcal{R}, b_{k'})^{k \in 1..m-1, k' \in m+1..l}$$

Store group:

$$\frac{\iota \notin dom(\sigma)}{(Group(\iota_k)^{k\in 1..l}, \sigma) \rightarrow_G (\iota, \{\iota \mapsto Gr(\iota_k)^{k\in 1..l}\} :: \sigma)}$$

Field access:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.l_i], \sigma) \rightarrow_G (Group(\iota_k.l_i)^{k\in 1..l}, \sigma)}$$

Field update:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.l_i := \iota'], \sigma) \rightarrow_G (Group(\iota_k.l_i := \iota')^{k\in 1..l}, \sigma)}$$

Invoke method:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.m_j(\iota')], \sigma) \rightarrow_G (Group(\iota_k.m_j(\iota'))^{k\in 1..l}, \sigma)}$$

Table 13.1. Reduction rules for groups

A Purely Syntactic Formulation

$$\begin{aligned} let\ Group = \lambda a_1 \ldots a_n.[\ & g_1 = a_1, \ldots, g_n = a_n, \\ & \forall j,\ m_j = \varsigma(x,y) Group(x.g_i.m_j(y))] \end{aligned}$$

13.2 Groups of Active Objects

$$ActiveGroup(a_1, \ldots, a_n, m) \triangleq Group(Active(a_1, m), \ldots, Active(a_n, m))$$

13.3 Groups, Determinism, and Atomicity

Atomicity

Confluence and Atomicity

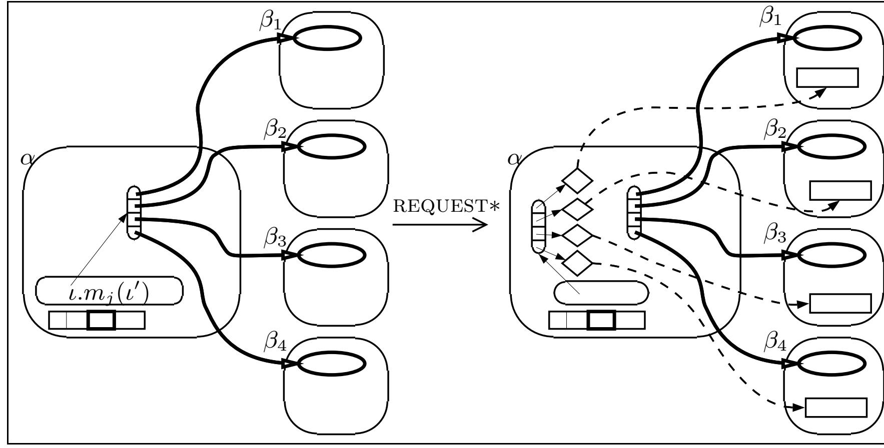


Fig. 13.2. Request sending to a group of active objects

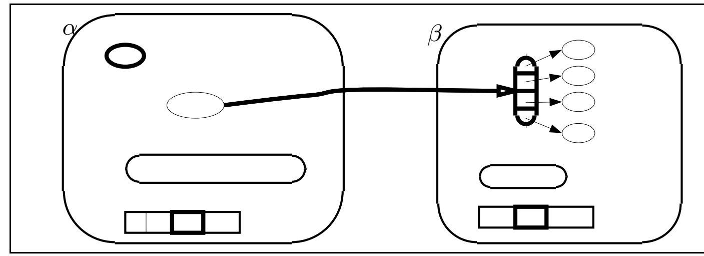


Fig. 13.3. An activated group of objects

Atomic group REQUEST:

$$\sigma(\iota) = Group(\iota_k)^{k \in 1..l}$$

$\forall k \in i..l$, if $\iota_k = AO(\beta_k)$ then

$$\alpha[\mathcal{R}[\iota_k.m_j(\iota')]; \sigma_{k-1}; \iota; F; R; f] \parallel P_{k-1} \xrightarrow{\text{REQUEST}_{\alpha\rightarrow\beta_k}} \alpha[\mathcal{R}[a_k]; \sigma_k; \iota; F; R; f] \parallel P_k$$

$$\text{else} \quad a_k = \iota_k.m_j(\iota') \ \wedge \ \sigma_k = \sigma_{k-1} \ \wedge \ P_k = P_{k-1}$$

$$\overline{\alpha[\mathcal{R}[\iota.m_j(\iota')]; \sigma_\alpha; \iota; F; R; f] \parallel P_0 \longrightarrow \alpha[\mathcal{R}[Group(a_k)^{k \in 1..l}]; \sigma_l; \iota; F; R; f] \parallel P_l}$$

Table 13.2. Atomic reduction rules for groups

$$
\begin{array}{l}
\textit{let } \beta_1 = [\ldots\ foo_1 = \varsigma(s)(\alpha_2.bar();\ldots), foo_2 = \varsigma(s)\ \ldots] \\
\qquad \ldots \\
\textit{and } \beta_4 = [\ldots\ foo_1 = \varsigma(s)\ldots, foo_2 = \varsigma(s)\ \ldots] \\
\textit{and } betas = ActiveGroup(\beta_1, \beta_2, \beta_3, \beta_4) \\
\textit{and } \alpha_1 = Active([\ldots\ main = \varsigma(s) betas.foo_1()\ldots]) \\
\textit{and } \alpha_2 = Active([\ldots\ bar = \varsigma(s) betas.foo_2()\ldots])\ \ \textit{in} \\
\qquad \alpha_1.main()
\end{array}
$$

Fig. 13.4. A confluent program if communications are atomic

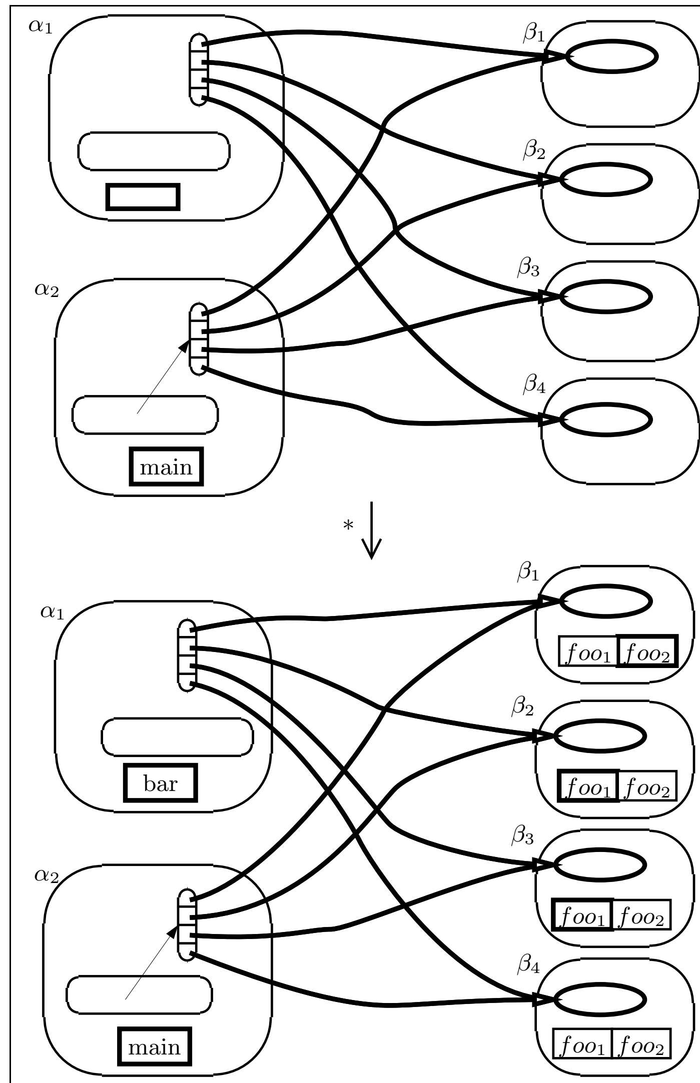


Fig. 13.5. Execution with atomic group communications

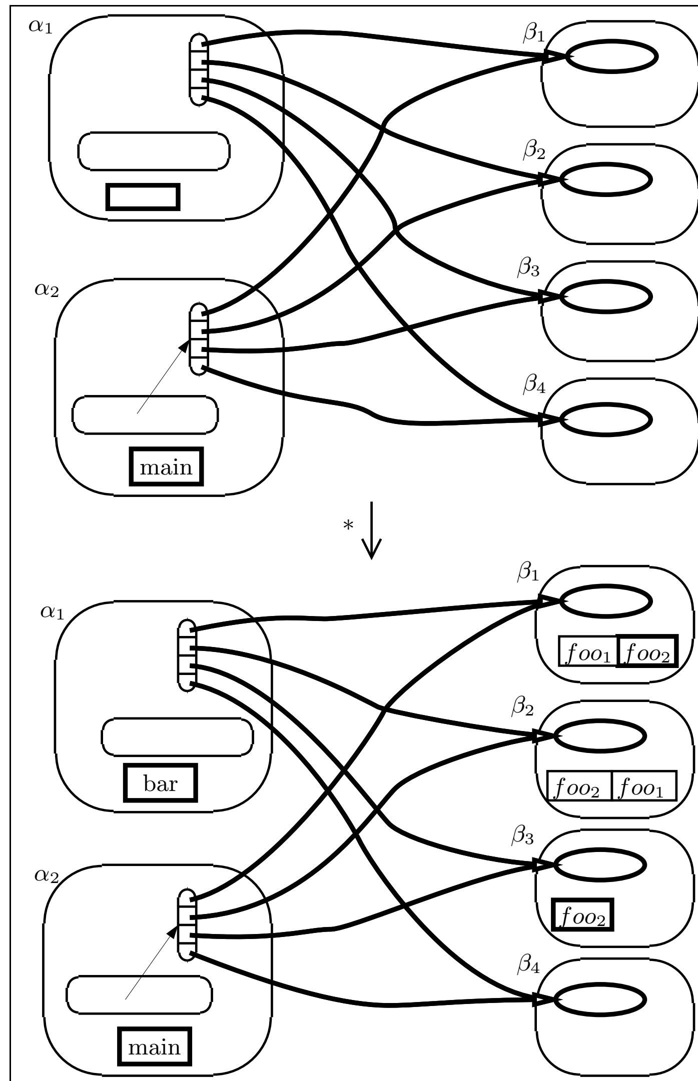


Fig. 13.6. An execution without atomic group communications

14

Components

14.1 From Objects to Components

Definition 14.1 (Primitive component)
A primitive component *is defined from an activity* α *(a root active object and a set of passive objects), a set of* Server Interfaces *(SIs), and a set of* Client Interfaces *(CIs).*

Each server interface SI_i *is a subset of the served methods.*

$$SI_i \subseteq \bigcup_{M \in \mathcal{M}_{\alpha P_0}} M$$

Served methods that do not belong to an interface correspond to asynchronous calls that can only be internal to a component.

A client interface (CI_j) *is a reference to an(other) activity contained in any attribute (field) of the active object:* $CI_j = l_i$ *where* l_i *is a field of the active object. More formally, we define a primitive component as follows:*

$$PC ::= C_n < a, srv, \{SI_i\}^{i \in 1..k}, \{CI_j\}^{j \in 1..l} >$$

where a *is an ASP term corresponding to the object to be activated and its dependencies (passive objects),* srv *is the service method of the object* a*, each* SI_i *is a set of method labels (as defined above), and each* CI_j *is a field name of the root object defined by* a*,* C_n *is the name of the defined component.*

14.2 Hierarchical Components

Definition 14.2 (Composite component)
A composite component is a set of components (either primitive (PC) or composite (CC)) exporting some server interfaces (some SI_i*), some client interfaces (some* CI_j*), and connecting some client and server interfaces (defining*

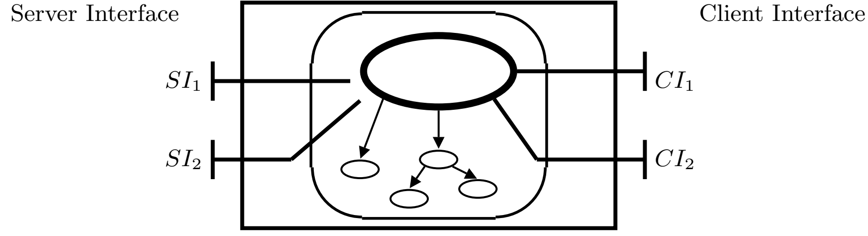


Fig. 14.1. A primitive component

a partial binding (CI_i, SI_j)*). Such a component is given a name* C_n*.* CC *is a composite component and* C *either a primitive or a composite one:*

$$CC ::= C_n \ll \quad C_1, \ldots, C_m; \{(C_{i_p}.CI_{j_p}, C_{i'_p}.SI_{j'_p})\}^{p \in 1..k}; \\ \{C_{i_q}.CI_{j_q} \to CI_q\}^{q \in 1..l}; \{C_{i_r}.SI_{j_r} \to SI_r\}^{r \in 1..l'} \gg$$

$$C ::= PC \mid CC$$

where each C_i *is the name of one included component* C_i $(i \in 1..m)$*, supposed to be pairwise distinct; each exported* SI *is only bound once to an included component, and each internal client interface* $(C_i.CI_j)$ *appears at most one time:*

$$\forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \begin{cases} p \neq p' \Rightarrow C_{i_p}.CI_{j_p} \neq C_{i_{p'}}.CI_{j_{p'}} \\ q \neq q' \Rightarrow C_{i_q}.CI_{j_q} \neq C_{i_{q'}}.CI_{j_{q'}} \\ C_{i_p}.CI_{j_p} \neq C_{i_q}.CI_{j_q} \\ r \neq r' \Rightarrow SI_r \neq SI_{r'} \end{cases}$$

14.3 Semantics

$$getSI \; : \; C.SI \times C \; \to \; C$$
$$getSI(C.SI_j, PC) = C$$
$$\forall CC = C \ll C_1, \ldots C_i \ldots, C_n; \ldots; \ldots; \{..C_i.SI_{j'} \to SI_j..\} \gg$$
$$getSI(C.SI_j, CC) = getSI(C_i.SI_{j'}, C_i)$$

$$getCI \; : \; C.CI \times C \; \to \; \mathcal{P}(C.CI)$$
$$getCI(C.CI_j, PC) = C.CI_j$$
$$\forall CC = C \ll C_1, \ldots, C_n; \ldots; ExportedCI; \ldots \gg$$
$$getCI(C.CI_j, CC) = \{getCI(C_i.CI_{j'}, C_i) | C_i.CI_{j'} \to CI_j \in ExportedCI\}$$

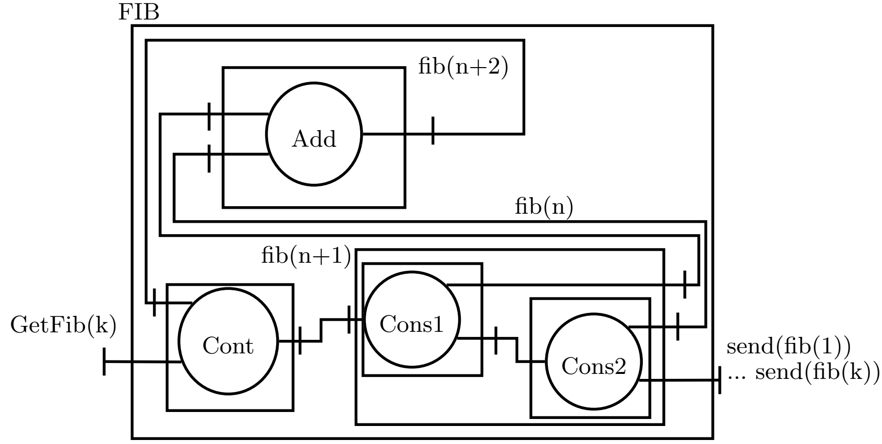


Fig. 14.2. Fibonacci as a composite component

$$Bind\ :\ C\ \rightarrow\ \mathcal{P}(C.CI{\times}C.SI)$$
$$Bind(PC) = \emptyset$$
$$\forall CC = C \ll C_1, .., C_n; Bindingss; ..; .. \gg$$
$$Bind(CC) = \bigcup_{\substack{(C_{i_p}.CI_{j_p}, C_{i'_p}.SI_{j'_p}) \in Bindings \\ C.CI \in getCI(C_{i_p}.CI_{j_p}, C_{i_p})}} \left\{ \left(C.CI, getSI(C_{i'_p}.SI_{j'_p}, C_{i'_p}) \right) \right\}$$
$$\cup \bigcup_{i \in 1..n} Bind(C_i)$$

$$PC_{i_k} = C_{i_k} < a_k, srv_k, \ldots, \ldots >^{k \in 1..N}$$

$$(C_p.CI_j, C_j) \in Bind(CC)\ \Rightarrow\ \exists q,\ (CI_j = CI_{pq} \wedge c(p,q) = j)$$

14.4 Deterministic Components

Definition 14.3 (Deterministic Primitive Component (DPC))
A DPC is a primitive component defined from an activity α. It verifies that server interfaces SIs are disjoint subsets of the served method of the active object of α such that every $M \in \mathcal{M}_{\alpha P_0}$ is (partially) included in a single SI_i.

$$\begin{cases} \forall i, k,\ i \neq k \Rightarrow SI_i \cap SI_k = \emptyset \\ \forall M \in \mathcal{M}_{\alpha P_0},\ \forall M_1 \subseteq M,\ \forall M_2 \subseteq M\ (M_1 \subseteq SI_i \wedge M_2 \subseteq SI_j) \Rightarrow i = j \end{cases}$$

$$SI_i = \bigcup_{\text{for some } M_j \in \mathcal{M}_{\alpha P_0}} M_j$$

$$
\begin{array}{l}
C_{Fibo} \ll C_{add} <[n1 = 0, n2 = 0, Cont = [];\\
\qquad\qquad srv = \varsigma(s)Repeat(Serve(set1); Serve(set2);\\
\qquad\qquad\qquad s.Cont.snd(s.n1 + s.n2)),\\
\qquad\qquad set1 = \varsigma(s,n)s.n1 := n, set2 = \varsigma(s,n)s.n2 := n],\\
\qquad\quad srv,\\
\qquad\quad \{\{set1\}, \{set2\}\},\\
\qquad\quad \{Cont\}\ \ >,\\
\quad C_{cont} <[CI = [];\\
\qquad\quad srv = \varsigma(s)Serve(GetFib),\\
\qquad\quad GetFib = \varsigma(s,k)Repeat_k(Serve(snd)),\\
\qquad\quad snd = \varsigma(s,k)s.CI.snd(k)],\\
\qquad srv,\\
\qquad \{\{GetFib\}, \{snd\}\},\\
\qquad \{CI\}\ \ >,\\
\quad CC \ll C_{cons1} <[add = [], out = [];\\
\qquad\qquad srv = \varsigma(s)\ (s.add.set1(1); s.out.snd(1); Repeat(Serve(snd)))\\
\qquad\qquad snd = \varsigma(s,n)(s.add.set1(n); s.out.snd(n))],\\
\qquad\quad srv,\\
\qquad\quad \{\{snd\}\},\\
\qquad\quad \{add, out\}\ \ >,\\
\qquad C_{cons2} < [add = [], out = [];\\
\qquad\qquad srv = \varsigma(s)\ (s.add.set2(0); s.out.snd(0); Repeat(Serve(snd)))\\
\qquad\qquad snd = \varsigma(s,n)(s.add.set2(n); s.out.snd(n))],\\
\qquad\quad srv,\\
\qquad\quad \{\{snd\}\},\\
\qquad\quad \{add, out\}\ \ >;\\
\qquad \{C_{cons1}.out \rightarrow C_{cons2}.\{snd\}\};\\
\qquad \{C_{cons2}.out \rightarrow Displ, C_{cons1}.add \rightarrow Add1, C_{cons2}.add \rightarrow Add2\};\\
\qquad \{C_{cons1}.\{snd\} \rightarrow In\} \gg;\\
\quad \{CC.Add1 \rightarrow C_{add}.\{set1\}, CC.Add2 \rightarrow C_{add}.\{set2\},\\
\qquad\quad C_{add}.Cont \rightarrow C_{cont}.\{snd\}, C_{cont}.CI \rightarrow CC.In\};\\
\quad \{CC.Displ \rightarrow Display\};\\
\quad \{C_{cont}.GetFib \rightarrow GetFib\} \gg
\end{array}
$$

Fig. 14.3. A definition of Fibonacci components

$$
\begin{array}{l}
let\ c_{i_1} = Active(((a_1.CI_{11} := c_{c(1,1)}). \ldots).CI_{1n_1} := c_{c(1,n_1)}, srv_1)\\
and\ \ldots\\
and\ c_{i_k} = Active(((a_k.CI_{k1} := c_{c(k,1)}). \ldots).CI_{kn_k} := c_{c(k,n_k)}, srv_k)\\
and\ \ldots\\
and\ c_{i_k} = Active(((a_N.CI_{N1} := c_{c(N,1)}). \ldots).CI_{kn_N} := c_{(N,n_N)}, srv_N)
\end{array}
$$

Fig. 14.4. Deployment of a composite component

Definition 14.4 (Deterministic Composite Component (DCC))
A DCC is

- *either a DPC,*
- *or a composite component connecting some DCCs such that the binding between server and client interfaces is one to one. More precisely the following constraints must be added to the ones of Definition 14.2:*

$$\begin{cases} \text{Each } C_i \text{ is a DCC} \\ \forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \begin{cases} p \neq p' \Rightarrow C_{i'_p}.SI_{j'_p} \neq C_{i'_{p'}}.SI_{j'_{p'}} \\ r \neq r' \Rightarrow C_{i_r}.SI_{j_r} \neq C_{i_{r'}}.SI_{j_{r'}} \\ C_{i'_p}.SI_{j'_p} \neq C_{i_r}.SI_{j_r} \\ q \neq q' \Rightarrow CI_q \neq CI_{q'} \end{cases} \end{cases}$$

Property 14.5 (DCC determinism)
DCC components behave deterministically.

14.5 Components and Groups: Parallel Components

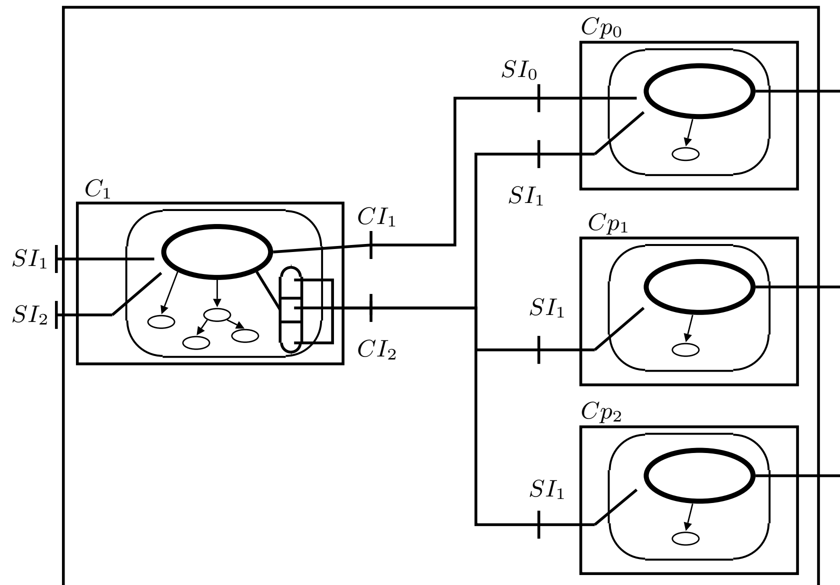


Fig. 14.5. A parallel component using groups

$$\ll \ldots ; \ldots , (C_1.CI_2, Cp_0.SI_1 \wedge Cp_1.SI_1 \wedge Cp_2.SI_1), \ldots ; \ldots ; \ldots \gg$$

$$\ll \ldots , Cp < \ldots > N, \ldots ; \ldots , (C_1.CI_1, Cp.SI_1), \ldots ; \ldots ; \ldots \gg$$

14.6 Components and Futures

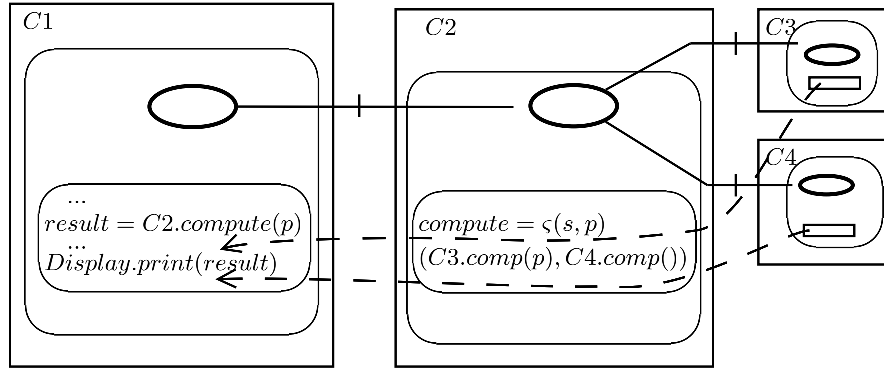


Fig. 14.6. Components and futures

15

Channels and Reconfigurations

15.1 Genuine ASP Channels

$$\forall i, j,\ i \neq j \Rightarrow \mathcal{C}_i^\alpha \cap \mathcal{C}_j^\alpha = \emptyset \quad \wedge \quad \forall M \in \mathcal{M}_{\alpha P_0},\ \exists i,\ M \subseteq \mathcal{C}_i^\alpha$$

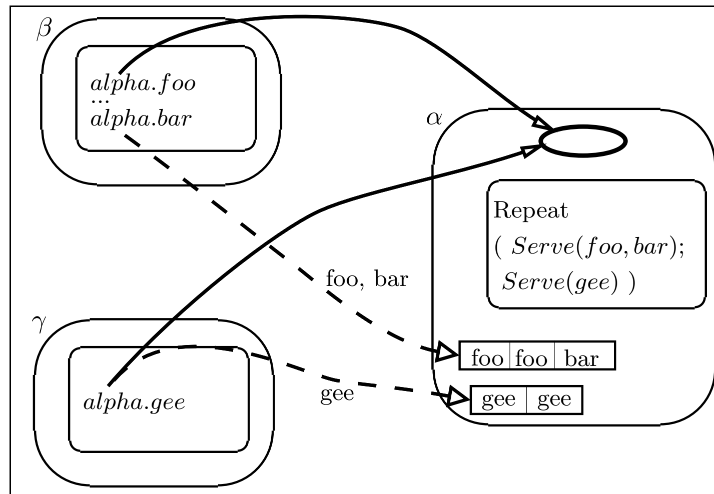


Fig. 15.1. Requests on separate channels do not interfere

15.2 Process Network Channels in ASP

$$\begin{aligned} Channel \triangleq [&value = [];activity = \varsigma(s)Repeat(Serve(put); Serve(get)), \\ &put = \varsigma(s, val)s.value := val, \quad get = \varsigma(s)s.value] \end{aligned}$$

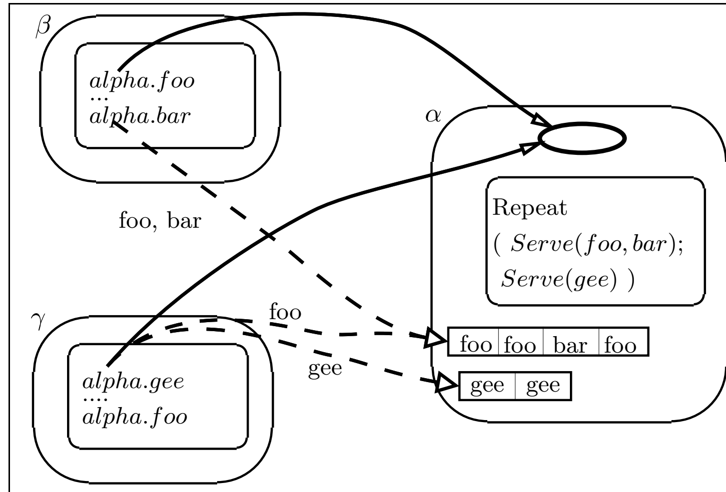


Fig. 15.2. A non-deterministic merge

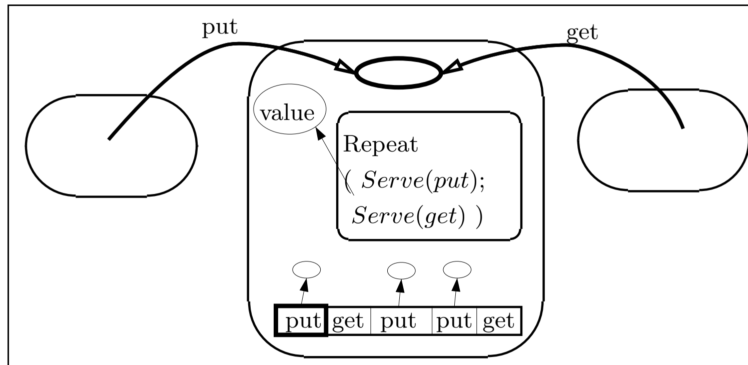


Fig. 15.3. A channel specified with an active object

15.3 Internal Reconfiguration

15.4 Event-Based Reconfiguration

Part V

Implementation Strategies

16

A Java API for ASP: ProActive

16.1 Design and API

16.1.1 Basic API and ASP Equivalence

ASP constructors	Java ProActive API
factory = Active(clone(a))	`ProActive.newActive("A", null, Node);`
Active(a, m)	`ProActive.turnActive(obj, Node);`
Serve(M)	`service.blockingServeOldest(String methodName);`
ServeWithoutBlocking(M)	`service.ServeOldest(String methodName);`
inQueue(M)	`service.hasRequestToServe (String methodName);`
waitFor(a)	`ProActive.waitFor(Object);`
awaited (a)	`ProActive.isAwaited(Object);`

Table 16.1. Relations between ASP constructors and ProActive API

16.1.2 Mapping Active Objects to JVMs: Nodes

```
A a = (A) ProActive.newActive("A",params,"rmi://lo.inria.fr/node");
```

```
A a = (A) ProActive.newActive("A", params, VirtualNodeRenderer);
```

16.1.3 Basic Patterns for Using Active Objects

```
A a = (A) ProActive.newActive("A", params, Node1);
```

ProActive static primitive	Migration toward
`migrateTo (Node)`	the JVM identified by the node
`migrateTo (Object)`	the current location of another active object

Table 16.2. Migration primitives in ProActive

```
v = a.bar (...) ;    // Asynchronous call, no wait
o.gee (v);           // No wait, even if o is a remote
                     // active object and v still awaited
...
v.f (...) ;          // Wait-by-necessity: wait until v
                     // gets its value
```

16.1.4 Migration

16.1.5 Group Communications

```
Object [][] params = {{...},...,{...}}; // An array of constructor params

A ag = (A) ProActiveGroup.newGroup("A", params,{node1,...,node2});
                              // A group of type "A" is created:
                              // member AOs are created at once
                              // on the specified nodes

ag.foo (...) ; // A group communication

V vg = ag.bar(); // A method call on a group with result vg:
...              // vg is a typed group of "V",
vg.f(); // Also a collective operation, subject to
        // wait-by-necessity
```

16.2 Examples

16.2.1 Parallel Binary Tree

16.2.2 Eratosthenes

```
public class SimpleAgent implements Serializable {

  public void moveToNode(Node n) {        // Move to a given node
      ProActive.migrateTo(n);
          logger.error("You should never see this, report a bug,");
          logger.error("or ... strong migration has been implemented!");
  }

  public void joinFriend(Object friend) {   // Move to join another AO
      ProActive.migrateTo(friend);
  }

  public String whereAreYou () {            // Repplies to queries
      return ("I am at " + InetAddress.getLocalHost ());
       ...
  }
                                    // Send information to other agents
  public void sendBestPrice (int myNewBestPrice) {
       ...
          myPeers[i].receiveBestPrice (myNewBestPrice);
       ...
  }
                                    // Receive information from other agents
  public void receiveBestPrice (int newBestPrice) {
      if (newBestPrice < currentBestPrice)
          currentBestPrice = newBestPrice;
  }
   ...
}
```

Fig. 16.1. A simple mobile agent in ProActive

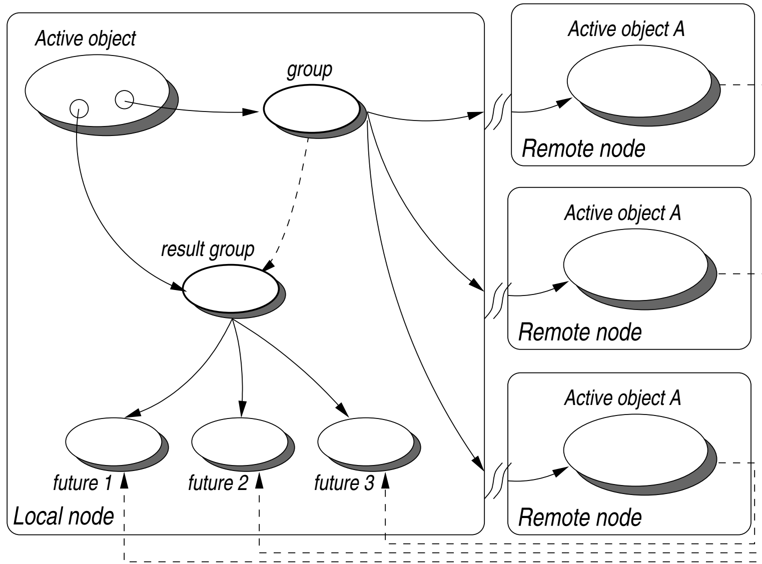


Fig. 16.2. Method call on group

```
        // Definition of one standard Java object and two active objects
        // Suppose B extends A
A a1 = new A();
A a2 = (A) ProActive.newActive("A", paramsA[], node);
B b  = (B) ProActive.newActive("B", paramsB[], node);

        // Creation of a group of active objects
Object [][] params  = {{...},...,{...}};
A ag1 = (A) ProActiveGroup.newGroup("A", params,{node1,...,node2});

        // An asynchronous one-way group communication
ag1.foo (...) ;
        // An asynchronous group communication with a group as a result
        // vg is a typed group of "V"
V vg = ag1.bar();
  ...
vg.f() ; // Also a collective operation, subject to wait-by-necessity


        // For management purposes, get the group view from the typed view
Group gA = ProActiveGroup.getGroup(ag1);

        // Now, add objects to the group:
        // active and non-active objects can be mixed in the group
gA.add(a1);
gA.add(a2);
gA.add(b);

        // The addition of members to a group immediately reflects
        // on the typed group view (ag1)
        // A method invoked on ag1 will also be called on a1, a2, b
ag1.foo() ;
V vg2 = ag1.bar();

        // A new reference to the typed group can also be built
A ag2 = (A) gA.getGroupByType();
```

Fig. 16.3. Dynamic typed group of active objects

```
public class BinaryTree} {
  protected int key;                    // The key contained in this object
  protected Object value;               // The value contained in this object
  protected BinaryTree leftTree;        // Left sub−tree
  protected BinaryTree rightTree; // Right sub−tree

  protected boolean isLeaf;          // Convenience instance variable
  // Class invariant:
  // (this . isLeaf == ((this. leftTree ==null) && (this.rightTree==null)))

  public BinaryTree() {              /* Constructor: creates an empty object */}
    this. isLeaf = true;  // On creation, an object has no child
  }
    /* Inserts a (key, value) pair in current sub−tree */}
  public void put (int key, Object value) {
    if (this. isLeaf) {                    // Object This is empty, let's use it
      this.key = key; this.value = value;
      this. isLeaf = false; this.createChildren();
    } else if (key == this.key) {      // Replaces the current value
      this.value = value;
    } else if (key < this.key) {        // Smaller keys are on the left
      this. leftTree .put(key, value);
    } else {                                 // Greater keys on the right
      this.rightTree.put(key, value);
    }
  }
    /* Returns the value associated to a given key */}
  public ObjectWrapper get (int key) {
    if (this. isLeaf) {                    // Reached a leaf, key not found
      return new ObjectWrapper ("null");
    }
    if (key == this.key) {              // Found the object, return value
      return new ObjectWrapper(this.value);
    }
    if (key < this.key) {                // Continue left, smaller keys
      ObjectWrapper res = this.leftTree.get(key);
      return res;
    }                                         // Continue right, greater keys
    ObjectWrapper res = this.rightTree.get(key);
    return res;
  }

  protected void createChildren() {         /* Creates two empty children */}
    this. leftTree = new BinaryTree();
    this.rightTree = new BinaryTree();
  }
}
```

Fig. 16.4. Sequential binary tree in Java

```
public class ActiveBinaryTree extends BinaryTree {

  protected void createChildren() {
      String s = this.getClass().getName();
      this.leftTree =  (BinaryTree) ProActive.newActive(s, null, null);
      this.rightTree = (BinaryTree) ProActive.newActive(s, null, null);
  }
}
```

Fig. 16.5. Subclassing binary tree for a parallel version

```
package org.objectweb.proactive.examples.binarytree;

public class Client {

  public static void main (String[] args) {
    BinaryTree tree = null;
    if     ( ... Sequential Version ... )
      tree = new BinaryTree ();
    else // Parallel Version
      tree =(Tree) ProActive.newActive("ActiveBinaryTree", null, null);

      myTree.put(3, "4");
      myTree.put(2, "3");
      myTree.put(5, "6");
      myTree.put(7, "8");
      ObjectWrapper res1 = myTree.get(5);         // (2)
      ObjectWrapper res2 = myTree.get(3);
      myTree.put(1, res1);                        // (1)
       ...
    }
```

Fig. 16.6. Main binary tree program in ProActive

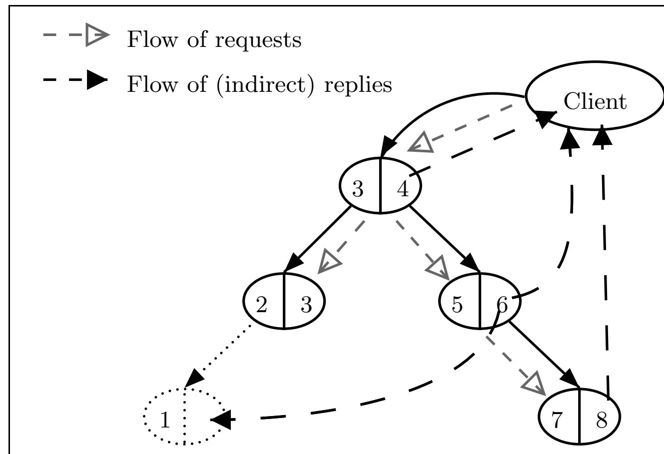


Fig. 16.7. Execution of the parallel binary tree program of Fig. 16.6

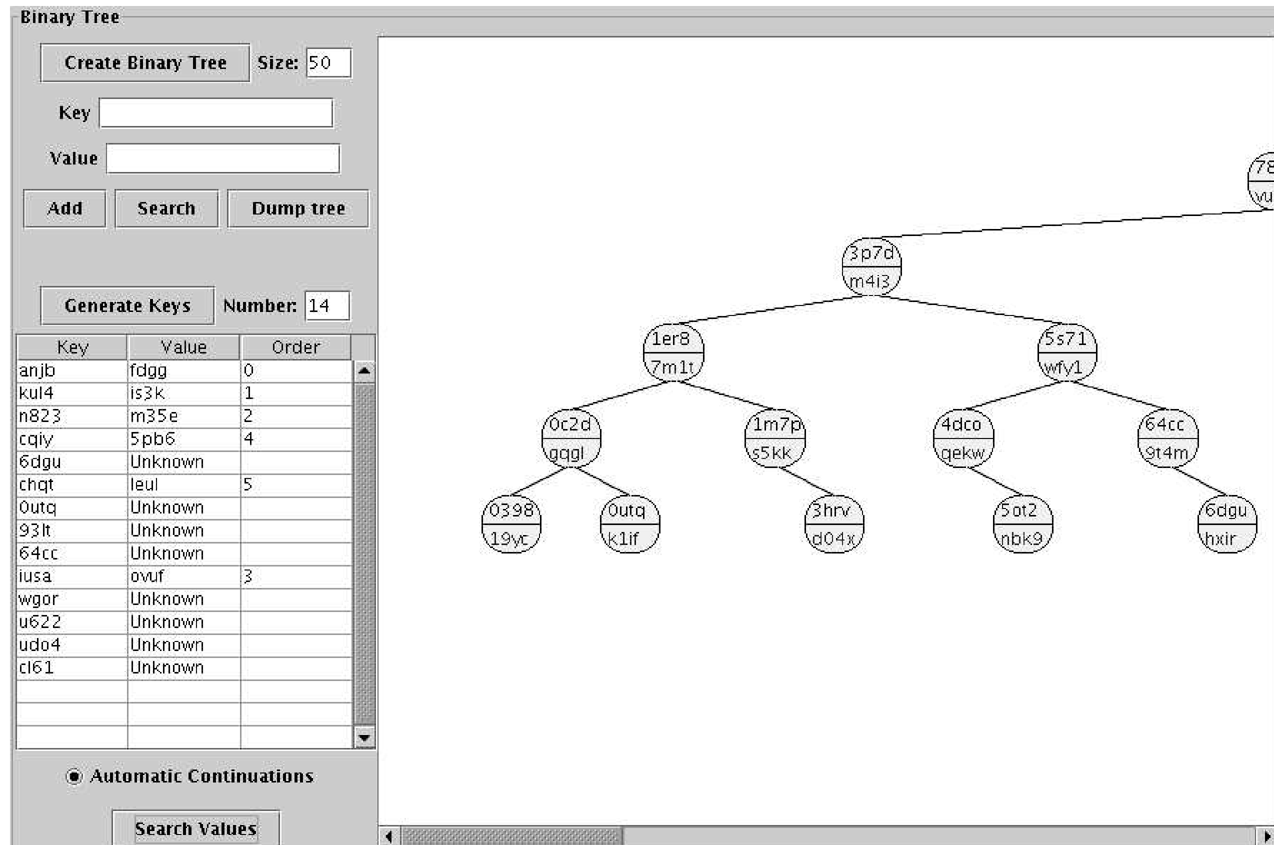


Fig. 16.8. Screenshot of the binary tree at execution

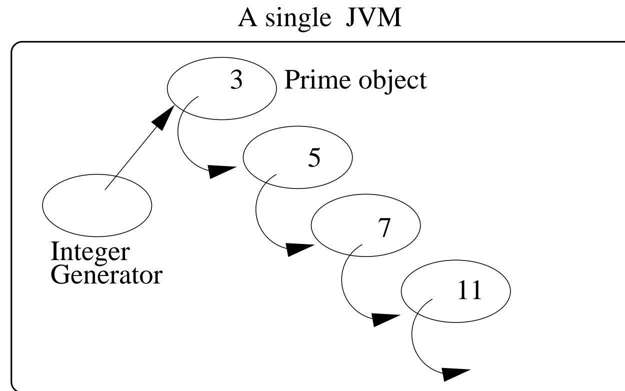


Fig. 16.9. Sequential Eratosthenes in Java

```
public class Prime {
  private long value;
  private Prime next;

  public Prime (long value) {                       /* Constructor */
     this.value = value;
  }
  public void isPrime (long n) {
      if (n % value == 0) // n is not prime, just drop it
         return;
      if (next == null) { // n is prime
        System.out.println ("New prime found: "+ n);
        next = createPrime (n);
      } else
        next.isPrime(n); // Don't know yet. Pass it to next prime object (1)
  }
  public Prime createPrime (long n) {
      return new Prime (n);
  }
}
```

Fig. 16.10. Sequential `Prime` Java class

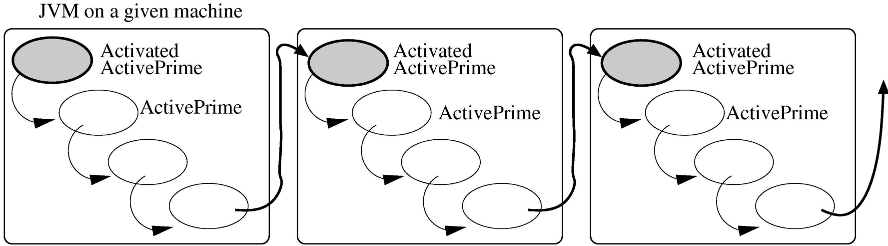


Fig. 16.11. Parallel Eratosthenes in Java ProActive

```
public class ActivePrime extends Prime {

  public ActivePrime (long value) {              /* Constructor */
     super(value);
  }
  public Prime createPrime (long n) {
    if ( timeToCreateANewActiveObject() )
      return (Prime)
         ProActive.newActive ("ActivePrime", {n}, nextAvailableNode());
    else
      return new ActivePrime (n);
  }
  public boolean timeToCreateANewActiveObject() {
     ... // Decide statically or dynamically if there are enough prime objects
     ... // in the current activity
  }
  public Node nextAvailableNode () {
     ... //Get a new or underloaded ProActive Node (machine+JVM),
     ... //where to locate the new activity
  }

}
```

Fig. 16.12. Parallel `ActivePrime` class

16.2.3 Fibonacci

```
if n <= 2 fib(n) = 1
if n > 2 fib(n) = fib(n-1) + fib(n-2)
```

```
service.blockingServeOldest("setFibN_2");
service.blockingServeOldest("setFibN_1");
cons1.setFibN(fibN_1.add(fibN_2));

service.barrier("setFibN_1", "setFibN_2");
cons1.setFibN(fibN_1.add(fibN_2));
```

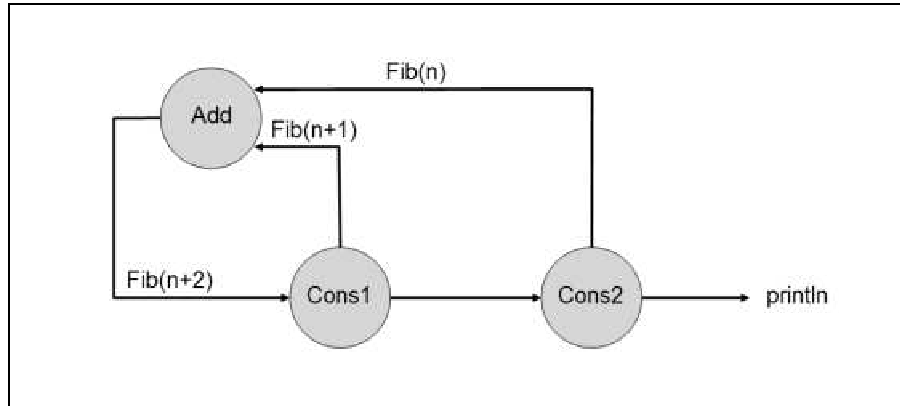


Fig. 16.13. Graph of active objects in the Fibonacci program

```
public static void main(String[] args) {
try {
  Add add = (Add) ProActive.newActive(Add.class.getName(), null);
  Cons1 cons1 =(Cons1) ProActive.newActive (Cons1.class.getName(),null);
  Cons2 cons2 =(Cons2) ProActive.newActive (Cons2.class.getName(),null);
  add.setCons1(cons1);
  cons1.setAdd(add);
  cons1.setCons2(cons2);
  cons2.setAdd(add);
}
      catch (ActiveObjectCreationException e) {
          e.printStackTrace();
      } catch (NodeException e) {
          e.printStackTrace();
      }
}
```

Fig. 16.14. Main Fibonacci program in ProActive

```
public class Add implements Serializable, InitActive, RunActive {
 private Cons1 cons1;
 private BigInteger fibN_1;
 private BigInteger fibN_2;

 public Add() { //Empty noArg constructor
 }
 public void initActivity (Body body) {
        Service service = new Service(body);
        service.blockingServeOldest("setCons1");
 }
 public void runActivity (Body body) {
        Service service = new Service(body);
        while (body.isActive()) {
             service.blockingServeOldest("setFibN_1");
             service.blockingServeOldest("setFibN_2");
             cons1.setFibN(fibN_1.add(fibN_2));
        }
 }
 public void setCons1 (Cons1 cons1) {
        this.cons1 = cons1;
 }
 public void setFibN_1 (BigInteger fibN_1) {
        this.fibN_1 = fibN_1;
 }
 public void setFibN_2 (BigInteger fibN_2) {
        this.fibN_2 = fibN_2;
 }
}
```

Fig. 16.15. The class `Add` of the Fibonacci program

```
public class Cons1 implements Serializable, InitActive, RunActive {
 private Add add;
 private Cons2 cons2;
 private BigInteger fibN;

 public Cons1() { //Empty no arg constructor
 }
 public void initActivity(Body body) {
        Service service = new Service (body);
        service.blockingServeOldest ("setAdd");
        service.blockingServeOldest ("setCons2");
 }
 public void runActivity(Body body) {
        Service service = new Service(body);
        add.setFibN_1 (BigInteger.ONE);
        cons2.setFibN_1 (BigInteger.ONE);
        while (body.isActive()) {
            service.blockingServeOldest ("setFibN");
            add.setFibN_1 (fibN);
            cons2.setFibN_1 (fibN);
        }
 }
 public void setAdd (Add add) {
        this.add = add;
 }
 public void setCons2 (Cons2 cons2) {
        this.cons2 = cons2;
 }
 public void setFibN (BigInteger fibN) {
                this.fibN = fibN;
 }
}
```

Fig. 16.16. The class `Cons1` of the Fibonacci program

```
public class Cons2 implements InitActive, RunActive {
 private Add add;
 private BigInteger fibN_1;

 public Cons2() { //Empty no arg constructor
 }
 public void initActivity (Body body) {
         Service service = new Service(body);
         service.blockingServeOldest("setAdd");
 }
 public void runActivity (Body body) {
         int k=0;
         Service service = new Service(body);
         add.setFibN_2(BigInteger.ZERO); // starting with 0
         k++;
         while (body.isActive()) {
              service.blockingServeOldest("setFibN_1");
              add.setFibN_2(fibN_1);
              System.out.println("Fib(" + k + ") = " + fibN_1);
              k++;
         }
 }
 public void setFibN_1 (BigInteger fibN_1) {
         this.fibN_1 = fibN_1;
 }
 public void setAdd (Add add) {
         this.add = add;
 }
}
```

Fig. 16.17. The class `Cons2` of the Fibonacci program

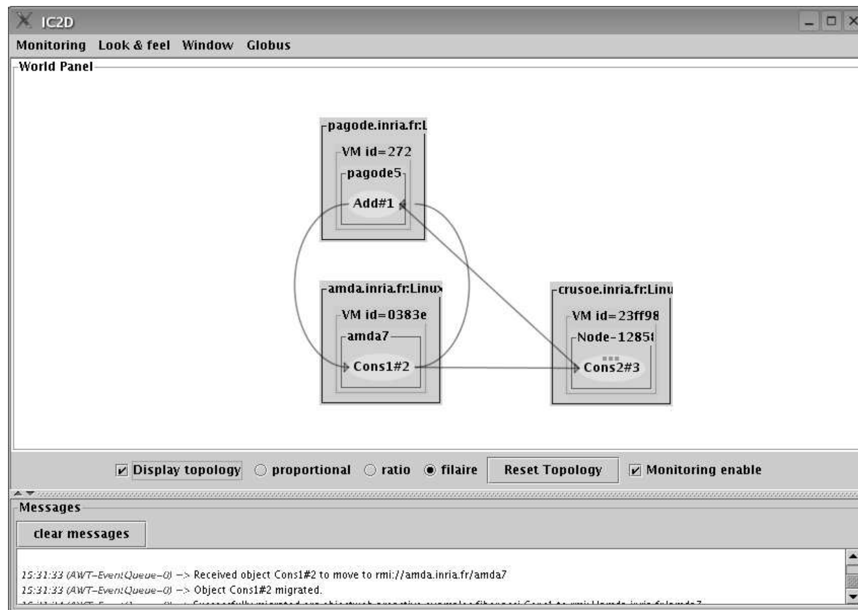


Fig. 16.18. Graphical visualization of the Fibonacci program using IC2D

17

Future Update

17.1 Future Forwarding

$$FF ::= \{(f_i^{\alpha\rightarrow\beta},\, \gamma,\, \delta)\}$$

with $(f_i^{\alpha\rightarrow\beta},\, \gamma,\, \delta) \in FF$ if the future reference $f_i^{\alpha\rightarrow\beta}$ has been transmitted from γ to δ.

Definition 17.1 (Forwarded futures)
In REQUEST *(sent from α to β),* $\quad f_i^{\gamma\rightarrow\delta} \in copy(\iota', \sigma_\alpha) \Rightarrow (f_i^{\gamma\rightarrow\delta}, \alpha, \beta) \in FF$
In REPLY *(sent from β to α),* $\quad f_i^{\gamma\rightarrow\delta} \in copy(\iota_f, \sigma_\beta) \Rightarrow (f_i^{\gamma\rightarrow\delta}, \beta, \alpha) \in FF$

Property 17.2 (Origin of futures)

$$fut(f_i^{\gamma\rightarrow\delta}) \in \sigma_\alpha \Rightarrow \alpha = \gamma \vee \exists\beta,\, (f_i^{\gamma\rightarrow\delta}, \beta, \alpha) \in FF$$

where the implication becomes an equivalence as soon as no garbage collection of future references is performed.

Property 17.3 (Forwarded futures flow)

$$(f_i^{\gamma\rightarrow\delta}, \beta, \alpha) \in FF \Rightarrow \beta = \gamma \vee \exists\beta',\, (f_i^{\gamma\rightarrow\delta}, \beta', \beta) \in FF$$

17.2 Update Strategies

17.2.1 ASP and Generalization: Encompassing All Strategies

17.2.2 No Partial Replies and Requests

Property 17.4 (No forwarded futures)

$$FF = \emptyset$$

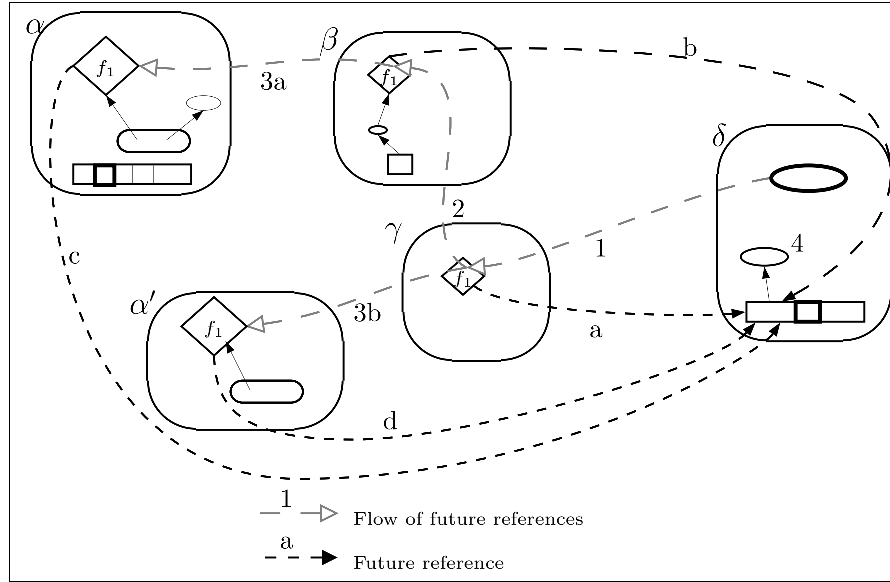


Fig. 17.1. A future flow example.

Generalized REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\delta}) \quad F_\beta(f_i^{\gamma\to\delta}) = \iota_f \qquad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P}$$

INFORM:

$$\frac{f_i^{\gamma\to\delta} \notin F_\alpha \quad F_\beta(f_i^{\gamma\to\delta}) = \iota_f \qquad \iota'_f \notin dom(\sigma_\alpha) \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f \ ; \ \sigma_\alpha, \iota'_f)}{\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \alpha[a_\alpha; \sigma'_\alpha; \iota_\alpha; F_\alpha :: \{f_i^{\gamma\to\delta} \mapsto \iota'_f\}; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P}$$

Table 17.1. Generalized future update

ENDSERVICE, for future $f_i^{\gamma\to\delta}$ on activity $\delta \hookrightarrow$ REPLY$^{\delta\to\gamma}(f_i^{\gamma\to\delta})$

Table 17.2. No partial replies and requests protocol

17.2.3 Forward-Based

$$FF_\alpha = \{(f_i^{\gamma\to\delta}, \alpha, \beta') | \exists f_i, \gamma, \delta, \beta' (f_i^{\gamma\to\delta}, \alpha, \beta') \in FF\}$$

Property 17.5 (Forward-based future update is eager)
If there is no cycle of future then, for each calculated future $f_i^{\gamma\to\delta}$, the protocol of Table 17.3 terminates with $\forall\beta$, $f_i^{\gamma\to\delta} \notin FutureRefs(\beta)$.

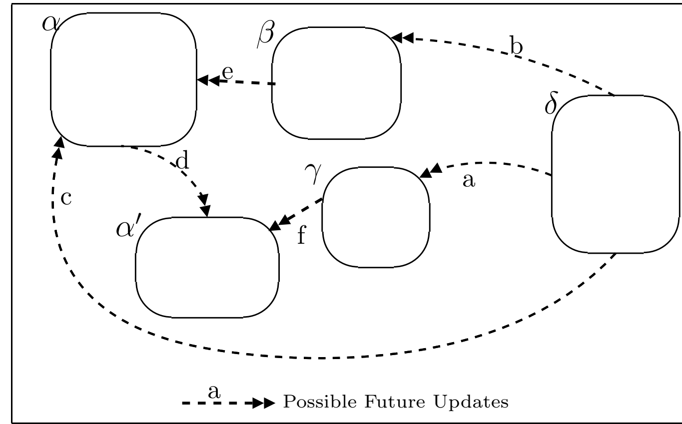


Fig. 17.2. General strategy: any future update can occur

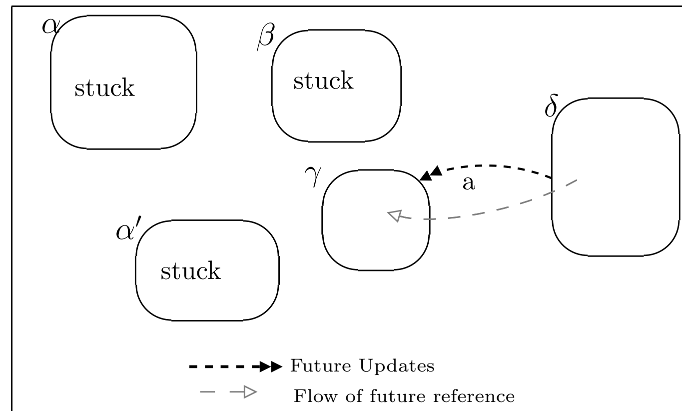


Fig. 17.3. No partial replies and requests

ENDSERVICE, for future $f_i^{\gamma\to\delta}$ on activity $\delta \hookrightarrow \begin{cases}\text{REPLY}^{\delta\to\gamma}(f_i^{\gamma\to\delta})\\ \text{INFORM}^{\delta\to\gamma}(f_i^{\gamma\to\delta})\end{cases}$
$\text{REPLY}^{\alpha\to\beta}(f_i^{\gamma\to\delta}) \hookrightarrow \forall\beta'$ s.t. $(f_i^{\gamma\to\delta},\beta,\beta')\in FF, \begin{cases}\text{REPLY}^{\beta\to\beta'}(f_i^{\gamma\to\delta})\\ \text{INFORM}^{\beta\to\beta'}(f_i^{\gamma\to\delta})\end{cases}$

Table 17.3. Forward-based protocol

17.2.4 Message-Based

$$FF_\alpha = \{(f_i^{\gamma\to\alpha},\beta,\beta')\mid\exists f_i,\gamma,\beta,\beta'(f_i^{\gamma\to\alpha},\beta,\beta')\in FF\}$$

Property 17.6 (Message-based strategy is eager)
If there is no cycle of futures then for each calculated future $f_i^{\gamma\to\delta}$, the algorithm of Fig. 17.4 terminates with $\forall\beta$, $f_i^{\gamma\to\delta}\notin FutureRefs(\beta)$.

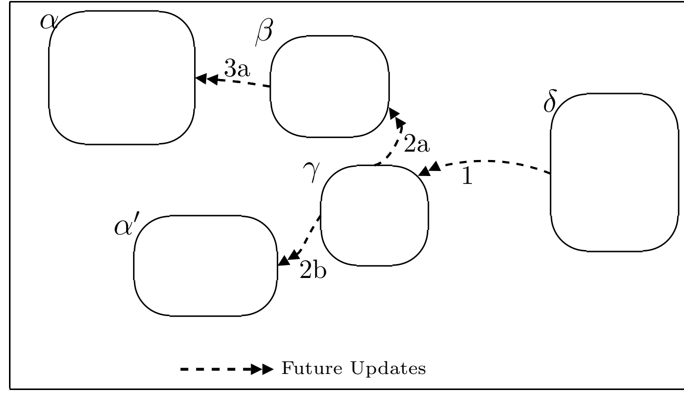


Fig. 17.4. Future updates for the forward-based strategy

ENDSERVICE, for future $f_i^{\gamma\to\delta}$ on activity $\delta \hookrightarrow$ $\begin{cases} \text{REPLY}^{\delta\to\gamma}(f_i^{\gamma\to\delta}) \\ \forall\beta' \text{ s.t. } \exists\beta(f_i^{\gamma\to\delta},\beta,\beta') \in FF,\ \text{REPLY}^{\delta\to\beta'}(f_i^{\gamma\to\delta}) \end{cases}$

Table 17.4. Message-based protocol for future update

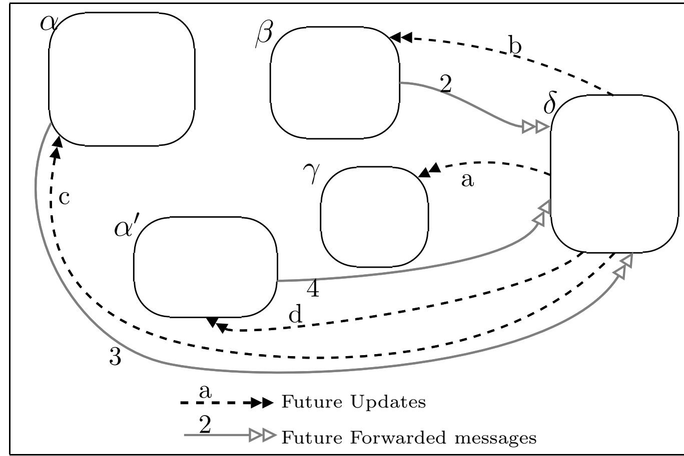


Fig. 17.5. Message-based strategy: future received and update messages

17.2.5 Lazy Future Update

REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta}) \quad F_\beta(f_i^{\gamma\to\beta}) = \iota_f \\ \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota) \quad \alpha \text{ is stuck by a wait-by-necessity}}{\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P}$$

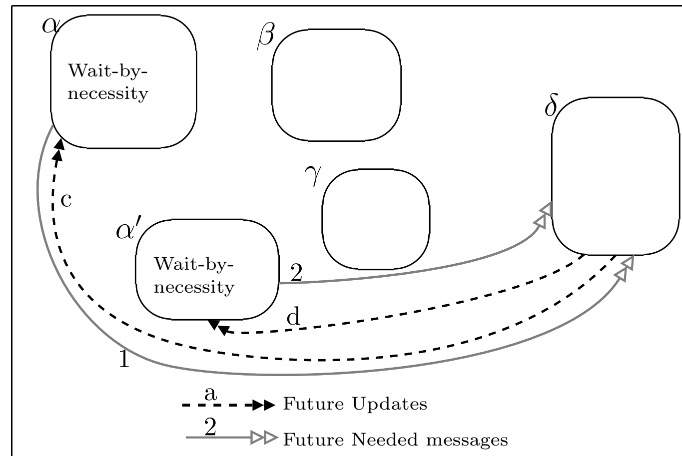


Fig. 17.6. Lazy future update: only needed futures are updated

17.3 Synthesis and Comparison of the Strategies

- *No partial replies and requests*: no passing of futures between activities, more deadlocks. Futures are not first-class entities.
- *Eager strategies*: as soon as a future is computed
  - *Forward-based*: each activity is responsible for updating the values of futures it has forwarded.
  - *Message-based*: each forwarding of a future generates a message sent to the activity that computes the future. The activity computing the future is responsible for sending its value to all.
- *Mixed strategy*: future updates can occur at any time between the future computation and the wait-by-necessity. For example, one could implement future updates on inactivity (any stuck configuration).
- *Lazy strategy*: on demand, only when the value of the future is needed (wait-by-necessity on *this* future).

Fig. 17.7. Future update strategies

18

Loosing Rendezvous

18.1 Objectives and Principles

$$
\begin{array}{l}
let\ \delta = Active([\ldots; m_j = \varsigma(s)\ldots])\ in \\
\quad let\ \beta = Active\big([;\ foo = \varsigma(s)\ \delta.m_2(); \\
\qquad\qquad\qquad\qquad Active([; act = \varsigma(s)\ \delta.m_3()], act); \\
\qquad\qquad\qquad\qquad \delta.m_4()\qquad], \emptyset\big)\ \ in \\
\qquad Active\big([; act = \varsigma(s)\ \delta.m_1(); \beta.foo(); \delta.m_5()], act\big)
\end{array}
$$

Fig. 18.1. Example: activities synchronized by rendezvous

- $m_1 :: m_2 :: m_3 :: m_4 :: m_5$, or
- $m_1 :: m_5 :: m_2 :: m_4 :: m_3$.

18.2 Asynchronous Without Guarantee

$$
\begin{array}{ll}
Active'(a) \triangleq & let\ act = Active(a, \emptyset)\ in \\
& let\ trRq = [; \forall m_j,\ m_j = \varsigma(s, arg)\ act.m_j(arg)]\ in \\
& \quad [; \forall m_j,\ m_j = \varsigma(s, arg)\ Active(trRq, \emptyset).m_j(arg)]
\end{array}
$$

where m_j ranges over the method labels of the activated object.

$$
\begin{array}{ll}
Active'(a, service) \triangleq & let\ act = Active(a, service)\ in \\
& let\ trRq = [; \forall m_j,\ m_j := \varsigma(s, arg)\ delegate(act.m_j(arg))]\ in \\
& \qquad [; \forall m_j,\ m_j := \varsigma(s, arg) Active(trRq, \emptyset).m_j(arg)]
\end{array}
$$

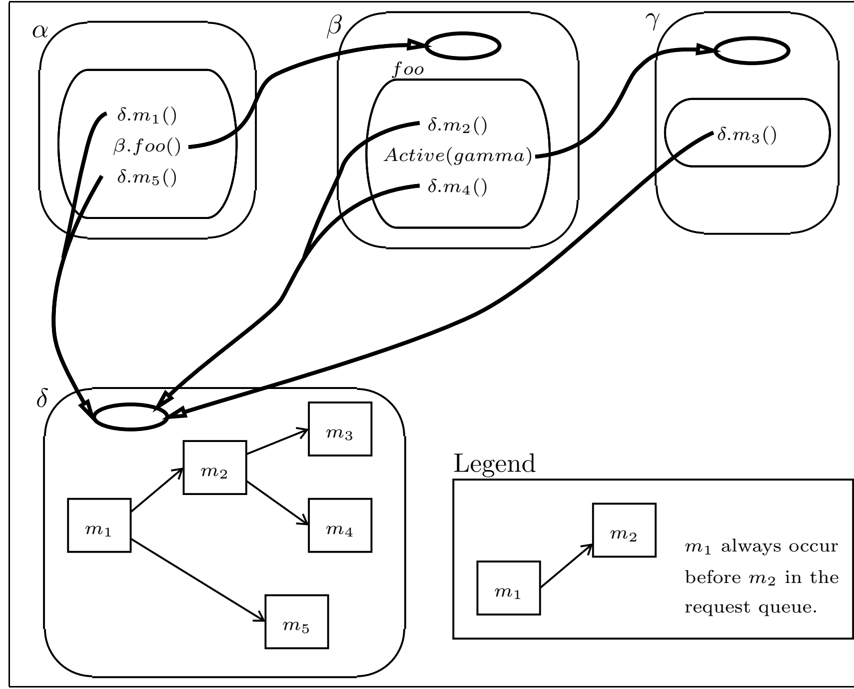


Fig. 18.2. ASP with rendezvous – message ordering

18.3 Asynchronous Point-to-Point FIFO Ordering

- $d \in Q$: existence of an entry,
- $Q\langle d\rangle$: find an object associated with an entry,
- $Q :: \{d \mapsto b\}$: add a new entry

$$queue \triangleq \varsigma(s, dest)\big(if\ (dst \notin s.Queues)\ then\ s.Queues := s.Queues :: \{dst \mapsto Active([;\forall m_j,\, m_j = \varsigma(s,z)\,\mathcal{D}(dst.m_j(z))], \emptyset)\}\big);\ s.Queues\langle dest\rangle$$

where $\mathcal{D}(a) = a$ if we consider the core ASP calculus, and $\mathcal{D}(a) = delegate(a)$ in the ASP calculus with delegation; m_j ranges over the method labels of the activated object.

$$Active'(a, service) \triangleq let\ act = Active(\mathcal{C}(a), service)\ in\ \big[;\forall m_j,\, m_j = \varsigma(s,y) thisActivity.queue(act).m_j(y)\big]$$

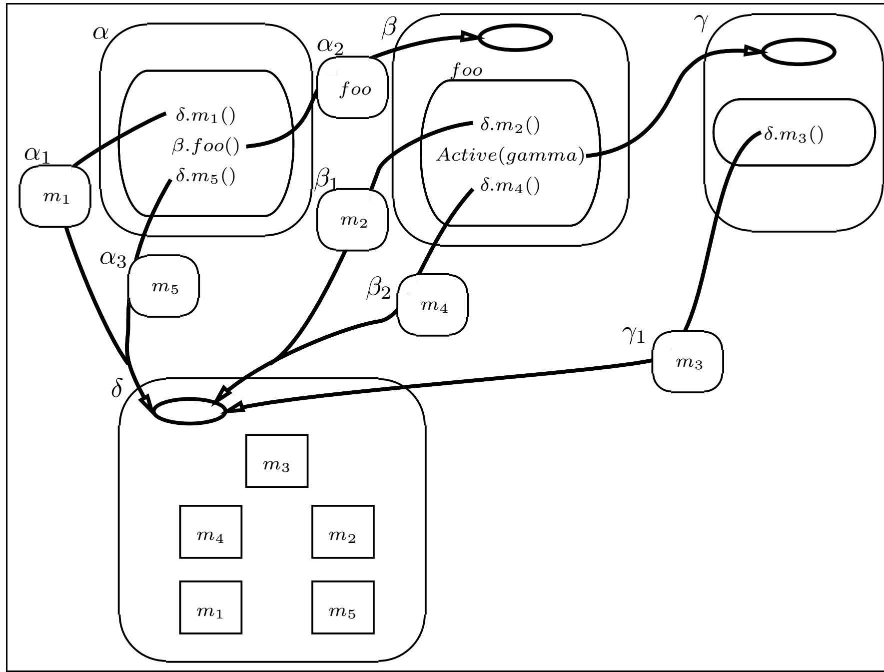


Fig. 18.3. Asynchronous communications without guarantee

Definition 18.1 (Triangle pattern)
A triangle pattern *is a configuration such that:*

$$\exists \alpha,\, \beta_1 \ldots \beta_n,\, \gamma \text{ s.t. } \alpha \rightarrow_R \gamma \text{ and then } \alpha \rightarrow_R \beta_1 \rightarrow_R \ldots \rightarrow_R \beta_n \rightarrow_R \gamma$$

where $\alpha \rightarrow_R \beta$ *means* α *sends requests to* β *or* α *creates activity* β*.*

Only request reception on the same channel is guaranteed to happen after a previously sent request has been received. But, compared to ASP with rendezvous, this loss of synchronization mainly has consequences for request reception order in the case of the triangle pattern.

- Execution is *insensitive to the moment when futures are updated*.
- Execution is characterized by the ordered list of request senders (*RSL confluence*).
- Request reception order is not fully guaranteed, e.g., in the case of the *triangle pattern.*

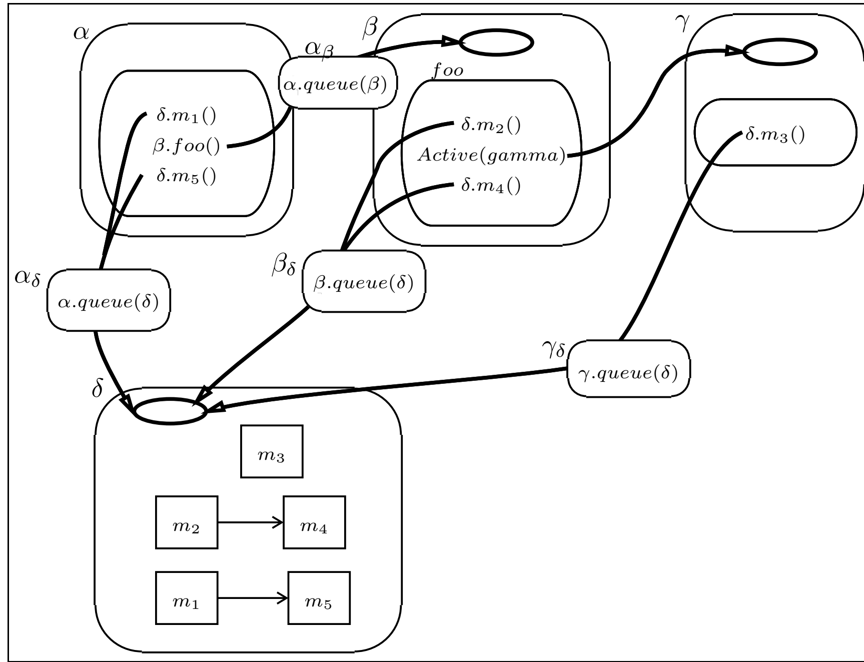


Fig. 18.4. Asynchronous point-to-point FIFO communications

18.4 Asynchronous One-to-All FIFO Ordering

$$\begin{aligned} Active'(a, service) \triangleq\ & let\ b = clone(a)\ \ in \\ & \Big(let\ Queue = [;\forall m_j \in \mathcal{M}_1,\ m_j = \varsigma(s,d,z)\,\mathcal{D}(d.m_j(z))]\ in \\ & \qquad b.queue := Active(Queue, \emptyset)\Big); \\ & let\ act = Active(b, service)\ in \\ & \quad \big[;\forall m_j \in \mathcal{M}_2,\ m_j = \varsigma(s,y)(thisActivity.queue).m_j(act,y)\big] \end{aligned}$$

$$\begin{aligned} Active'(a, service) \triangleq\ & let\ b = clone(a)\ \ in \\ & \Big(let\ Queue = [;\forall m_j,\ m_j = \varsigma(s,d,z)]\,\mathcal{D}(d.m_j(z)), \\ & \qquad\qquad QActive = \varsigma(s,x,srv)Active(x,srv)]\quad in \\ & \qquad b.queue := Active(Queue, \emptyset)\Big); \\ & let\ act = thisActivity.queue.QActive(b, service)\ in \\ & \quad \big[;\forall m_j,\ m_j = \varsigma(s,y)(thisActivity.queue).m_j(act,y)\big] \end{aligned}$$

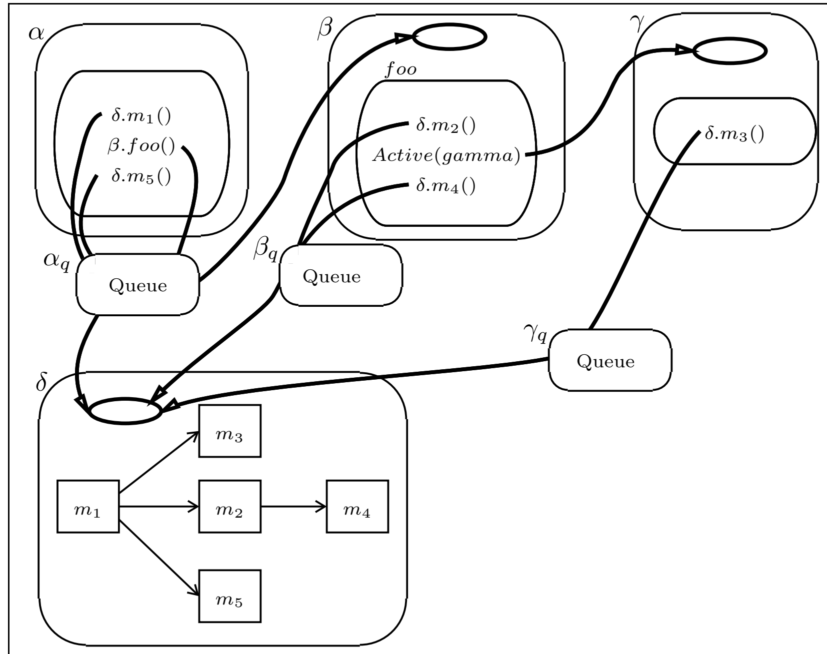


Fig. 18.5. Asynchronous one-to-all FIFO communications

18.5 Conclusion

19

Controlling Pipelining

- *Pure demand driven*: an activity is only activated when another one is waiting for data coming from it.
- *Limited pipelining*: the number of pending requests per sender activity is limited.
- *Unrestricted parallelism*: all activities are executed in parallel without any restriction.

Fig. 19.1. Strategies for controlling parallelism

19.1 Unrestricted Parallelism

19.2 Pure Demand Driven

19.3 Controlled Pipelining

20

Garbage Collection

20.1 Local Garbage Collection

20.2 Futures

20.3 Active Objects

Part VI

Final Words

21

ASP Versus Other Concurrent Calculi

21.1 Basic Formalisms

21.1.1 Actors

21.1.2 π-calculus and Related Calculi

$foo(bool\ b, bool\ c)$	
$\quad r = oa.m();$	gets a future
$\quad if\ b\ then\ r.bar;$	subject to wait-by-necessity
$\quad if\ c\ then\ r = [a = 1, b = 2];$	creates a local object
$\quad oa2.bar(r);$	sends to $oa2$ a possible future possibly awaited

21.1.3 Process Networks

Comparing ASP and Process Networks Channels

21.1.4 ς-calculus

21.2 Concurrent Calculi and Languages

21.2.1 MultiLisp

21.2.2 Ambient Calculus

21.2.3 join-calculus

21.3 Concurrent Object Calculi and Languages

21.3.1 Obliq and Øjeblik

21.3.2 The $\pi o\beta\lambda$ Language

22 Conclusion

22.1 Summary

22.2 A Dynamic Property for Determinism

22.3 ASP in Practice

22.4 Stateful Active Objects vs. Immutable Futures

	Nature	**Reference**	**Communication passing**	**Localization need due to**	**Localization strategy**
Active object	Object: stateful, mutable	Global	Reference semantics	Mobility	Forwarding Server TTL-TTU
Future	Method result: single assignment immutable	Global	Reference + Copy semantics	First-class futures Delegation	Eager: forward / message Mixed Lazy
Object	Stateful, mutable	Local	Copy	None	None

Table 22.1. Duality active objects (stateful) and futures (immutable)

22.5 Perspectives

23

Epilogue

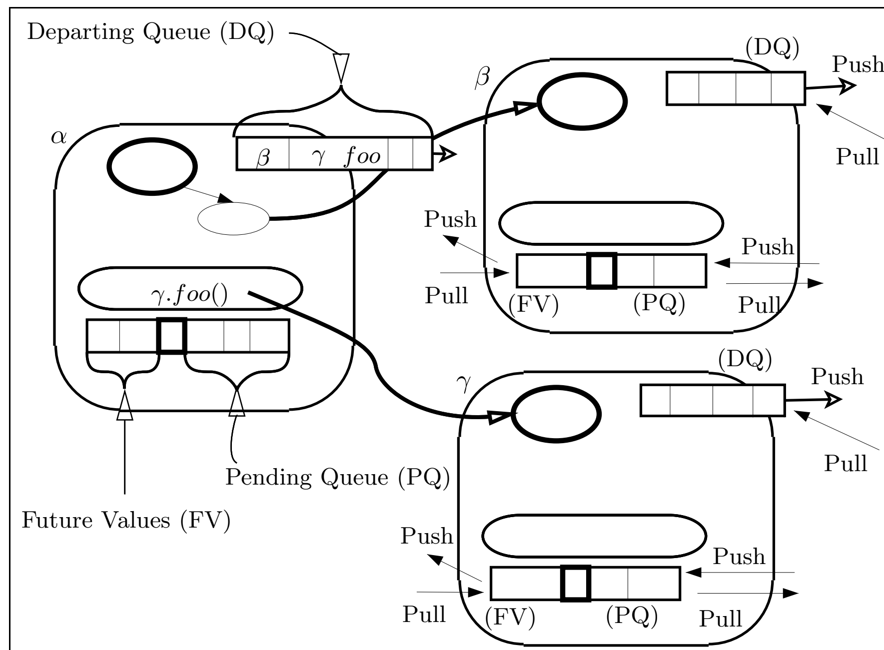


Fig. 23.1. Potential queues, buffering, pipelining, and strategies in ASP

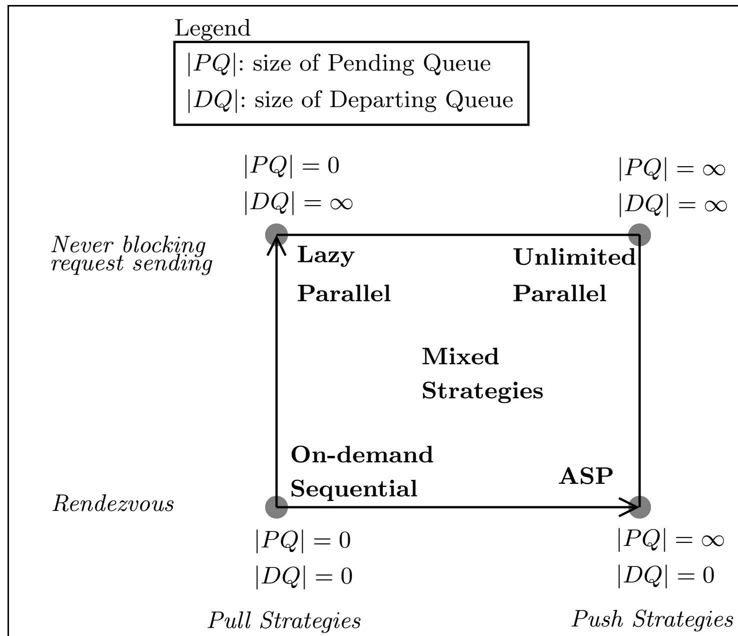


Fig. 23.2. Classification of strategies for sending requests

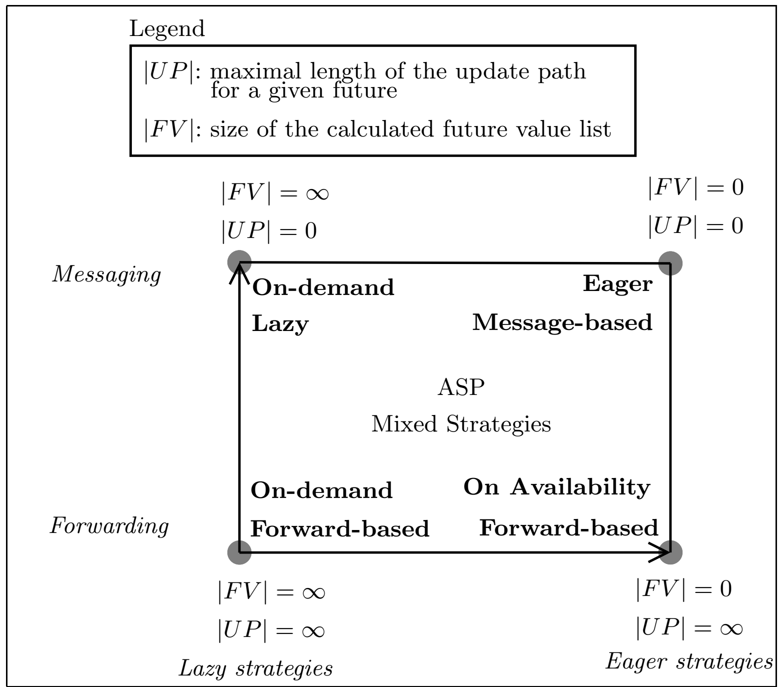


Fig. 23.3. Classification of strategies for future update

Appendices

A

Equivalence Modulo Future Updates

A.1 Renaming

$$\Theta ::= \{\!\!\{ f_i^{\beta \to \alpha} \leftarrow f_i'^{\beta \to \alpha}, \ldots \}\!\!\};$$

A.2 Reordering Requests ($R_1 \equiv_R R_2$)

$$\frac{\{M \in \mathcal{M}_{\alpha_{P_0}} | m_1 \in M\} \cap \{M \in \mathcal{M}_{\alpha_{P_0}} | m_2 \in M\} = \emptyset}{R_1 :: [m_1; \iota_1; f_1] :: [m_2; \iota_2; f_2] :: R_2 \equiv_R R_1 :: [m_2; \iota_2; f_2] :: [m_1; \iota_1; f_1] :: R_2}$$

$$R \equiv_R R$$

$$\frac{R \equiv_R R_1 \quad R_1 \equiv_R R'}{R \equiv_R R'}$$

Table A.1. Reordering requests

A.3 Future Updates

A.3.1 Following References and Sub-terms

Definition A.1 ($a \overset{\alpha}{\mapsto}_L b$)
$a \overset{\alpha}{\mapsto}_L b$ means that b is reached from a by following, inside the activity α, the

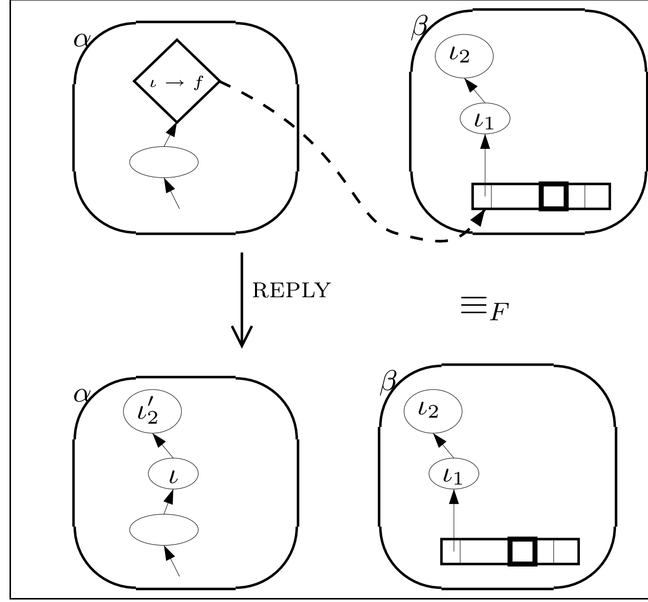


Fig. A.1. Simple example of future equivalence

path defined by L. Such paths are defined inductively from the definition of Table A.2 as follows:

$$a \overset{\alpha}{\mapsto}_{\emptyset} a$$
$$a \overset{\alpha}{\mapsto}_{L.x} b \text{ } if \text{ } a \overset{\alpha}{\mapsto}_{L} a' \wedge a' \overset{\alpha}{\mapsto}_{x} b$$

Extending the preceding definition, $\overset{\alpha}{\mapsto}_{L_1.L_2}$ denotes the concatenation $\overset{\alpha}{\mapsto}_{L_1}\overset{\alpha}{\mapsto}_{L_2}$. More generally the following notations are equivalent:

$$\begin{aligned} a \overset{\alpha}{\mapsto}_{L_1.L_2} b &\Leftrightarrow a \overset{\alpha}{\mapsto}_{L_1}\overset{\alpha}{\mapsto}_{L_2} b \\ &\Leftrightarrow \exists c,\, a \overset{\alpha}{\mapsto}_{L_1} c \overset{\alpha}{\mapsto}_{L_2} b \\ &\Leftrightarrow \exists c,\, a \overset{\alpha}{\mapsto}_{L_1} c \wedge c \overset{\alpha}{\mapsto}_{L_2} b \end{aligned}$$

Definition A.2 ($a \overset{\alpha*}{\mapsto}_L b$)

$$a \overset{\alpha*}{\mapsto}_{L_0 \ldots L_n} b \Leftrightarrow \begin{cases} (n = 0 \wedge a \overset{\alpha}{\mapsto}_{L_0} b) \quad \vee \\ \exists \iota_i,\, f_i,\, \beta_i,\, \gamma_i{}^{i \le n} \begin{cases} a \overset{\alpha}{\mapsto}_{L_0} fut(f_1^{\gamma_1 \to \beta_1}) \wedge F_{\beta_1}(f_1^{\gamma_1 \to \beta_1}) = \iota_1 \wedge \\ \sigma_{\beta_1}(\iota_1) \overset{\beta_1}{\mapsto}_{L_1} fut(f_2^{\gamma_2 \to \beta_2}) \wedge F_{\beta_2}(f_2^{\gamma_2 \to \beta_2}) = \iota_2 \\ \wedge \ldots \wedge \\ \sigma_{\beta_n}(\iota_n) \overset{\beta_n}{\mapsto}_{L_n} b \end{cases} \end{cases}$$

Lemma A.3 ($\overset{\alpha}{\mapsto}_L$ and $\overset{\alpha*}{\mapsto}_L$)

$$a \overset{\alpha}{\mapsto}_L b \Rightarrow a \overset{\alpha*}{\mapsto}_L b$$

$$\iota \overset{\alpha}{\mapsto}_{ref}\ \sigma_\alpha(\iota) \qquad [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \overset{\alpha}{\mapsto}_{l_i}\ b_i$$

$$[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \overset{\alpha}{\mapsto}_{m_j(s',x')}\ a_j\{\!\{x_j \leftarrow s', y_j \leftarrow x'\}\!\}$$

$$a.l_i \overset{\alpha}{\mapsto}_{field(l_i)}\ a \qquad a.l_i := b \overset{\alpha}{\mapsto}_{Update1(l_i)}\ a \qquad a.l_i := b \overset{\alpha}{\mapsto}_{Update2(l_i)}\ b$$

$$a.l_i(b) \overset{\alpha}{\mapsto}_{Invoke1(l_i)}\ a \qquad a.l_i(b) \overset{\alpha}{\mapsto}_{Invoke2(l_i)}\ b \qquad clone(a) \overset{\alpha}{\mapsto}_{clone}\ a$$

$$Active(a, m_j) \overset{\alpha}{\mapsto}_{Active(m_j)}\ a \qquad Serve(M) \overset{\alpha}{\mapsto}_{Serve(M)}\ \emptyset$$

$$a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow curr}\ a \qquad a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow f}\ a \qquad a \Uparrow f, b \overset{\alpha}{\mapsto}_{\Uparrow cont}\ b$$

$$\alpha_p \overset{\alpha}{\mapsto}_{ct}\ a_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{ao}\ \iota_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{cf}\ \iota_\alpha \qquad \alpha_p \overset{\alpha}{\mapsto}_{fv}\ F_\alpha$$

$$\frac{R_\alpha \equiv_R R'}{\alpha_p \overset{\alpha}{\mapsto}_{rq}\ R'}$$

$$[m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs_meth}\ m \qquad [m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs_arg}\ \iota$$

$$[m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs_fut}\ f \qquad [m; \iota; f] :: R \overset{\alpha}{\mapsto}_{reqs_cdr}\ R$$

$$\{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs_id}\ f_i^{\gamma\to\alpha} \qquad \{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs_val}\ \iota$$

$$\{f_i^{\gamma\to\alpha} \mapsto \iota\} :: F \overset{\alpha}{\mapsto}_{futs_cdr}\ F$$

Table A.2. Path definition

Lemma A.4 (Uniqueness of path destination)

$$a \overset{\alpha*}{\mapsto}_L b \wedge a \overset{\alpha*}{\mapsto}_L b' \Rightarrow b = b' \vee \exists f_n, \iota_n, \beta_n, \gamma, \delta \begin{cases} (\sigma_{\beta_n}(\iota_n) = fut(f_n^{\gamma\to\delta}) \vee F_\delta(f_n^{\gamma\to\delta}) = \iota_n) \\ \wedge(b = \iota_n \vee b' = \iota_n) \end{cases}$$

$$a \overset{\alpha*}{\mapsto}_L b \wedge a \overset{\alpha*}{\mapsto}_L b' \Rightarrow b = b' \vee \exists \iota_n, \iota'_n, \begin{cases} (\sigma_{\beta_n}(\iota_n) = fut(f_n^{\gamma\to\delta}) \wedge F_\delta(f_n^{\gamma\to\delta}) = \iota'_n) \\ (b = \iota_n \wedge b' = \iota'_n) \vee (b = \iota'_n \wedge b' = \iota_n) \end{cases}$$

A.3.2 Equivalence Definition

Definition A.5 (Equivalence modulo future updates: $P \equiv_F Q$)

$$P \equiv_F Q \Leftrightarrow \forall \alpha \ s.t. \ \alpha \in P \vee \alpha \in R, \forall L \left(\exists a, \alpha_P \overset{\alpha*}{\mapsto}_L a \Leftrightarrow \exists a', \alpha_R \overset{\alpha*}{\mapsto}_L a'\right) \quad (1)$$

$$\wedge \forall L, L', a \left(\alpha_P \overset{\alpha}{\mapsto}_L a \wedge \alpha_P \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L'_0, a', L_1, L'_1, \gamma \begin{cases} L = L_0.L_1 \wedge L' = L'_0.L'_1 \wedge L_1 \neq \emptyset \\ \alpha_R \overset{\alpha*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} a' \\ \wedge \alpha_R \overset{\alpha*}{\mapsto}_{L'_0} c \overset{\gamma}{\mapsto}_{L'_1} a' \end{cases} \right) \quad (2)$$

$$\wedge \forall L, L' \, a \left(\alpha_R \overset{\alpha}{\mapsto}_L a \wedge \alpha_R \overset{\alpha}{\mapsto}_{L'} a \Rightarrow \exists c, L_0, L'_0, a', L_1, L'_1, \gamma \begin{cases} L = L_0.L_1 \wedge L' = L'_0.L'_1 \wedge L_1 \neq \emptyset \\ \alpha_P \overset{\alpha*}{\mapsto}_{L_0} c \wedge \alpha_P \overset{\alpha*}{\mapsto}_{L'_0} c \\ c \overset{\gamma}{\mapsto}_{L_1} a' \wedge c \overset{\gamma}{\mapsto}_{L'_1} a' \end{cases} \right) \quad (3)$$

where $R = Q\Theta$ and Θ is a renaming of future identifiers like those defined in Sect. A.1.

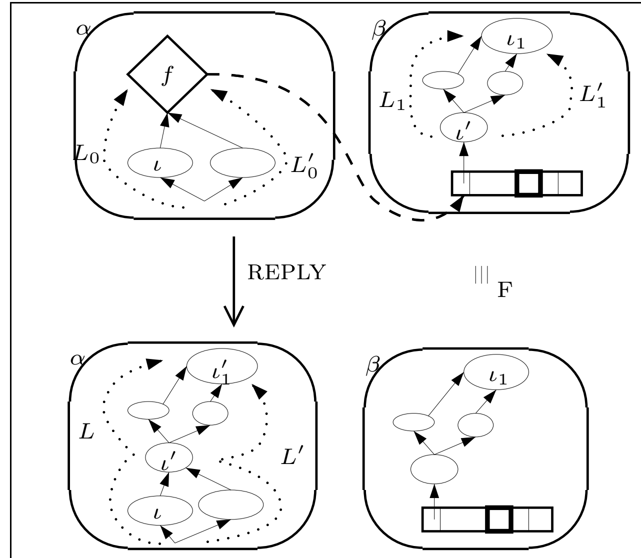


Fig. A.2. The principle of the alias conditions

Property A.6 (Equivalence relation)
$\equiv_F$ *is an equivalence relation.*

Definition A.7 (Equivalence of sub-terms)

$$a \equiv_F a' \Leftrightarrow \exists L,\, a \in \alpha_P \wedge a' \in \alpha_Q \wedge P \equiv_F Q \wedge \left(\alpha_P \overset{\alpha*}{\mapsto}_L a, \Leftrightarrow \alpha_Q \overset{\alpha*}{\mapsto}_L a'\right)$$

Lemma A.8 (Sub-term equivalence)

$$\begin{aligned} a \equiv_F a' \Rightarrow \forall L &\left(\exists b,\, a \overset{\alpha*}{\mapsto}_{L'} b \Leftrightarrow \exists b',\, a' \overset{\alpha*}{\mapsto}_{L'} b'\right) \\ &\wedge \forall L,\, L',\, b,\, a \overset{\alpha}{\mapsto}_L b \wedge a \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \exists c, L_0, L_0', b', L_1, L_1', \gamma \\ &\qquad\qquad \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset \\ a' \overset{\alpha*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} b' \\ a' \overset{\alpha*}{\mapsto}_{L_0'} c \overset{\gamma}{\mapsto}_{L_1'} b' \end{cases} \\ &\wedge \forall L,\, L',\, b,\, a' \overset{\alpha}{\mapsto}_L b \wedge a' \overset{\alpha}{\mapsto}_{L'} b \Rightarrow \exists c, L_0, b', L_1, L_1', \gamma \\ &\qquad\qquad \begin{cases} L = L_0.L_1 \wedge L' = L_0'.L_1' \wedge L_1 \neq \emptyset \\ a \overset{\alpha*}{\mapsto}_{L_0} c \overset{\gamma}{\mapsto}_{L_1} b' \\ a \overset{\alpha*}{\mapsto}_{L_0'} c \overset{\gamma}{\mapsto}_{L_1'} b' \end{cases} \end{aligned}$$

A.4 Properties of $\equiv_F$

Property A.9 (Equivalence and compatibility)

$$P \equiv_F Q \Rightarrow P \bowtie Q$$

Lemma A.10 ($\equiv_F$ and store update)

$$\begin{cases} P \equiv_F Q \quad \wedge \quad a \equiv_F a' \\ \iota \in dom(\sigma_{\alpha_P}) \wedge \iota' \in dom(\sigma_{\alpha_Q}) \quad \wedge \quad \iota \equiv_F \iota' \\ P' = P \ except\ \sigma_{\alpha_{P'}} = \{\iota \to a\} + \sigma_{\alpha_P} \\ Q' = Q \ except\ \sigma_{\alpha_{Q'}} = \{\iota' \to a'\} + \sigma_{\alpha_Q} \\ \vdash P' \text{ OK} \ \wedge \ \vdash Q' \text{ OK} \end{cases} \Rightarrow P' \equiv_F Q'$$

Lemma A.11 ($\equiv_F$ and substitution)

$$\iota \equiv_F \iota' \Rightarrow a\{\!\{x \leftarrow \iota\}\!\} \equiv_F a\{\!\{x \leftarrow \iota'\}\!\}$$

Lemma A.12 (A characterization of deep copy)

$$a \in copy(\iota, \sigma_\beta) \Leftrightarrow \exists L,\, \iota \overset{\beta}{\mapsto}_L a$$

$$\iota' \in dom(copy(\iota, \sigma)) \Rightarrow copy(\iota, \sigma)(\iota') = \sigma(\iota')$$

Lemma A.13 (Copy and merge)
If $P' = P \ except\ \sigma_{\alpha_{P'}} = Copy\&Merge(\sigma_{\beta_P}, \iota_0 \ ;\ \sigma_{\alpha_P}, \iota)$ *then*

$$\iota_0 \overset{\beta_P}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}}{\mapsto}_L a'$$

and

$$\iota_0 \overset{\beta_P*}{\mapsto}_L a \Leftrightarrow \iota \overset{\alpha_{P'}*}{\mapsto}_L a'$$

Lemma A.14 ($\equiv_F$ and store merge)

$$\begin{cases} P \equiv_F Q \;\wedge\; \iota \in \alpha_P \;\wedge\; \iota' \in \alpha_Q \wedge \iota_0 \in \beta_P \;\wedge\; \iota'_0 \in \beta_Q \\ a \equiv_F a' \quad \wedge \quad \iota \equiv_F \iota' \quad \wedge \quad \iota_0 \equiv_F \iota'_0 \\ P' = P \;\mathit{except}\; \sigma_{\alpha_{P'}} = \mathit{Copy\&Merge}(\sigma_{\beta_P}, \iota_0 \;;\; \sigma_{\alpha_P}, \iota) \\ Q' = Q \;\mathit{except}\; \sigma_{\alpha_{Q'}} = \mathit{Copy\&Merge}(\sigma_{\beta_Q}, \iota'_0 \;;\; \sigma_{\alpha_Q}, \iota') \end{cases} \Rightarrow P' \equiv_F Q'$$

A.5 Sufficient Conditions for Equivalence

Property A.15 (REPLY and $\equiv_F$)

$$P \xrightarrow{\text{REPLY}} P' \Rightarrow P \equiv_F P'$$

Property A.16 (Sufficient condition for equivalence)

$$\begin{cases} P_1 \xrightarrow{\text{REPLY}} P' \\ P_2 \xrightarrow{\text{REPLY}} P' \end{cases} \quad \Rightarrow \quad P_1 \equiv_F P_2$$

$$\begin{cases} P \xrightarrow{\text{REPLY}} P_1 \\ P \xrightarrow{\text{REPLY}} P_2 \end{cases} \quad \Rightarrow \quad P_1 \equiv_F P_2$$

A.6 Equivalence Modulo Future Updates and Reduction

Property A.17 ($\equiv_F$ and reduction(1))

$$P \equiv_F P' \wedge P \xrightarrow{T} Q \Rightarrow \begin{cases} \mathit{if}\; T = \text{REPLY} \;\mathit{then}\; Q \equiv_F P' \\ \mathit{else}\; \exists Q',\, P' \xrightarrow{\text{REPLY}*}\xrightarrow{T} Q' \wedge Q' \equiv_F Q \end{cases}$$

Definition A.18 (Parallel reduction modulo future updates)

$$\overset{T}{\Longrightarrow} \quad = \quad \begin{array}{ll} \xrightarrow{\text{REPLY}*}\xrightarrow{T} & \mathit{if}\; T \neq \text{REPLY} \\ \xrightarrow{\text{REPLY}*} & \mathit{if}\; T = \text{REPLY} \end{array}$$

Property A.19 ($\equiv_F$ and reduction(2))

$$P \equiv_F P' \wedge P \xrightarrow{T} Q \Rightarrow \exists Q',\, P' \overset{T}{\Longrightarrow} Q' \wedge Q' \equiv_F Q$$

Corollary A.20 ($\equiv_F$ and reduction)

$$P \equiv_F P' \wedge P \overset{T}{\Longrightarrow} Q \Rightarrow \exists Q',\, P' \overset{T}{\Longrightarrow} Q' \wedge Q' \equiv_F Q$$

A.7 Another Formulation

$$\Theta ::= (\theta_{fut}, \theta_{\alpha}^{\alpha\in P}, \ldots, \theta_{\alpha\to\beta,\iota}^{\alpha\in P,\,\beta\in Q}, \ldots)$$
$$\theta_{fut} ::= \{\!\{ f_i^{\beta_P\to\alpha_P} \leftarrow f_i'^{\beta_Q\to\alpha_Q}, \ldots \}\!\};$$

for $\alpha \in P \wedge \alpha \in Q$,

$$\theta_\alpha ::= \{\!\{ \iota_1 \leftarrow \iota_1' \ldots \}\!\} \text{ where } \iota_1 \in dom(\sigma_{\alpha_P}), \iota_1' \in dom(\sigma_{\alpha_Q})$$

for $\alpha_P \in P$, $\beta_Q \in Q$, $\iota \in dom(\sigma_{\alpha_P})$, $\iota' \in dom(\sigma_{\beta_Q})$,

$$\theta_{\alpha_P\to\beta_Q,\iota'} ::= \{\!\{ \iota_1 \leftarrow \iota_1', \ldots \}\!\} \text{ where } \iota_1 \in dom(\sigma_{\alpha_P}),\ \iota', \iota_1' \in dom(\sigma_{\beta_Q})$$

$$\theta_{\alpha_P\leftarrow\beta_Q,\iota} ::= \{\!\{ \iota_1 \leftarrow \iota_1', \ldots \}\!\} \text{ where } \iota, \iota_1 \in dom(\sigma_{\alpha_P}),\ \iota_1' \in dom(\sigma_{\beta_Q})$$

- for each $\beta \in Q$ the sets $codom(\theta_\beta)$, $(codom(\theta_{\alpha\to\beta,\iota'}))_{\alpha\in P,\iota\in dom(\sigma_\beta)}$ are disjuncts;
- for each $\alpha \in P$ the sets $dom(\theta_\alpha)$, $(dom(\theta_{\alpha\leftarrow\beta,\iota}))_{\beta\in Q,\iota\in dom(\sigma_\alpha)}$ are disjuncts.

Definition A.21 (Equivalence modulo future updates (2))
To prove $P \equiv_F Q$, one must provide

$$\Theta = (\theta_{fut}, \theta_{\alpha}^{\alpha\in P}, \ldots, \theta_{\alpha\to\beta,\iota}^{\alpha\in P,\,\beta\in Q}, \ldots)$$

such that the rules of Table A.3 are coinductively verified (for all activities α of P and Q).

In Table A.3, x replaces α or $\alpha \to \beta, \iota'$ or $\alpha \leftarrow \beta, \iota$. All the trivial induction rules corresponding to operators in the syntax are not given explicitly in this table.

$\equiv_\alpha$ denotes the equivalence between (sub-)terms that appears in α in both configurations. $\equiv_{\alpha\to\beta,\iota'}$ denotes the equivalence between terms contained in a future calculated in the activity α of P and its updated value in the location ι' of the activity β of Q. $\equiv_x$ denotes any equivalence relation (either inside an activity when $x = \alpha$ or between activities when x is of the form $\alpha \leftarrow \beta', \iota$ or $\alpha \to \beta', \iota'$).

Property A.22 (Equivalence of the two equivalence definitions) *The two definitions of the equivalence modulo future updates presented in this appendix are equivalent.*

A.8 Decidability of $\equiv_F$

Property A.23 (Decidability)
$\equiv_F$ is decidable.

$$fut(f) \equiv_x fut(f\theta_{fut}) \qquad AO(\alpha) \equiv_x AO(\alpha) \qquad \emptyset \equiv_x \emptyset$$

$$\frac{\sigma_{\alpha_P}(\iota) \equiv_\alpha \sigma_{\alpha_Q}(\iota\theta_\alpha)}{\iota \equiv_\alpha \iota\theta_\alpha}$$

$$\frac{\sigma_{\alpha_P}(\iota_1) \equiv_{\alpha\leftarrow\beta,\iota_0} \sigma_{\beta_Q}(\iota_1\theta_{\alpha\leftarrow\beta,\iota})}{\iota \equiv_{\alpha\leftarrow\beta,\iota_0} \iota\theta_{\alpha\leftarrow\beta,\iota}}$$

$$\frac{\begin{array}{c} b \text{ is in the location } \iota \text{ of the activity } \gamma \text{ of } P \\ F_{\beta_Q}(f_i^{\alpha\to\beta}) = \iota' \quad \iota \equiv_{\gamma\leftarrow\beta,\iota} \iota' \end{array}}{b \equiv_x fut(f_i^{\alpha\to\beta})}$$

$$\frac{\sigma_{\alpha_P}(\iota) \equiv_{\alpha\to\beta,\iota'_0} \sigma_{\beta_Q}(\iota\theta_{\alpha\to\beta,\iota'_0})}{\iota \equiv_{\alpha\to\beta,\iota'_0} \iota\theta_{\alpha\to\beta,\iota'_0}}$$

$$\frac{\begin{array}{c} a \text{ is in the location } \iota' \text{ of the activity } \gamma \text{ of } Q \\ F_{\beta_P}(f_i^{\alpha\to\beta}) = \iota \quad \iota \equiv_{\beta\to\gamma,\iota'} \iota' \end{array}}{fut(f_i^{\alpha\to\beta}) \equiv_x a}$$

$$\frac{\iota \equiv_\alpha \iota' \quad R_{\alpha_P} \equiv_\alpha R_{\alpha_Q}}{[m_j, \iota, f] :: R_{\alpha_P} \equiv_\alpha [m_j, \iota', f\theta_{fut}] :: R_{\alpha_Q}}$$

$$\frac{\iota \equiv_\alpha \iota' \quad F \equiv_\alpha F'}{\{f_i^{\gamma\to\alpha_P} \mapsto \iota\} :: F \equiv_\alpha \{f_i^{\gamma\to\alpha_Q}\theta_{fut} \mapsto \iota'\} :: F'}$$

$$\frac{\begin{array}{c} a_{\alpha_P} \equiv_\alpha a_{\alpha_Q} \quad \iota_{\alpha_P} \equiv_\alpha \iota_{\alpha_Q} \quad F_{\alpha_P} \equiv_\alpha F_{\alpha_Q} \\ \forall M \in \mathcal{M}_{\alpha_P},\, R_{\alpha_P}\big|_M \equiv_F R_{\alpha_Q}\big|_M \qquad f_{\alpha_Q} = f_{\alpha_P}\theta_{fut} \end{array}}{\alpha[a_{\alpha_P}; \sigma_{\alpha_P}; \iota_{\alpha_P}; F_{\alpha_P}; R_{\alpha_P}; f_{\alpha_P}] \equiv_\alpha \alpha[a_{\alpha_Q}; \sigma_{\alpha_Q}; \iota_{\alpha_Q}; F_{\alpha_Q}; R_{\alpha_Q}; f_{\alpha_Q}]}$$

$$\frac{\begin{array}{c} \exists\Theta = (\theta_{fut}, \theta_{\alpha_1}, \ldots) \quad \alpha \in P \Leftrightarrow \alpha \in Q \\ \forall\alpha \in P, \alpha[a_{\alpha_P}; \sigma_{\alpha_P}; \iota_{\alpha_P}; F_{\alpha_P}; R_{\alpha_P}; f_{\alpha_P}] \equiv_\alpha \alpha[a_{\alpha_Q}; \sigma_{\alpha_Q}; \iota_{\alpha_Q}; F_{\alpha_Q}; R_{\alpha_Q}; f_{\alpha_Q}] \end{array}}{P \equiv_F Q}$$

Table A.3. Equivalence rules

A.9 Examples

$$\theta_{\beta\to\alpha,\iota} = \{\!\{\iota_1 \leftarrow \iota, \iota_2 \leftarrow \iota'_2\}\!\}$$

$$\begin{aligned} \theta_{\alpha\to\alpha,\iota_2} &= \{\!\{\iota_1 \leftarrow \iota_2, \iota_2 \leftarrow \iota_3\}\!\} \\ \theta_{\alpha\to\alpha,\iota_3} &= \{\!\{\iota_1 \leftarrow \iota_3, \iota_2 \leftarrow \iota_4\}\!\} \\ \theta_{\alpha\to\alpha,\iota_4} &= \{\!\{\iota_1 \leftarrow \iota_4, \iota_2 \leftarrow \iota_5\}\!\} \end{aligned}$$

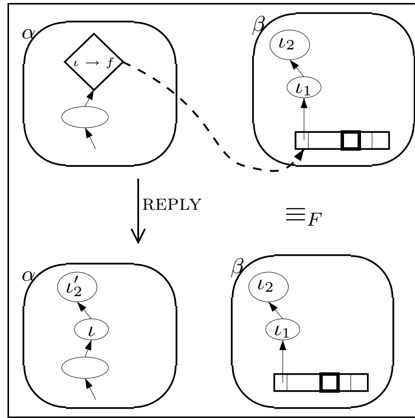


Fig. A.3. Simple example of future equivalence

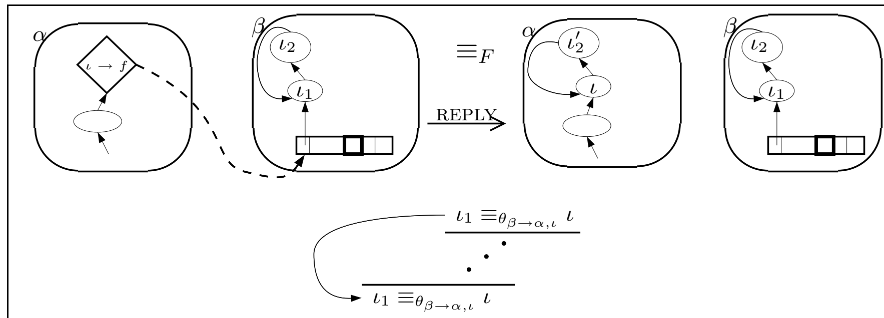


Fig. A.4. Example of a "cyclic" proof

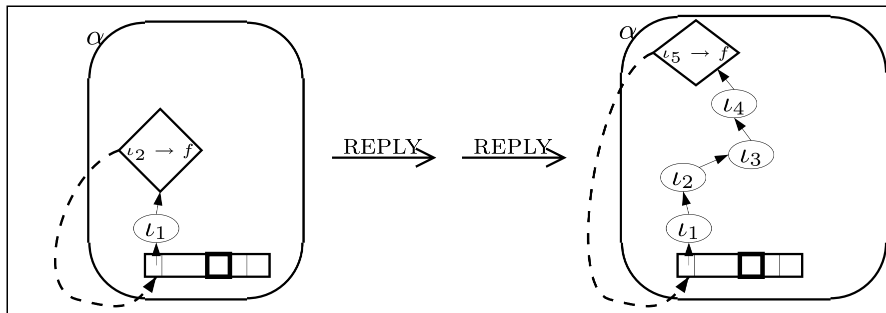


Fig. A.5. Equivalence in the case of a cycle of futures

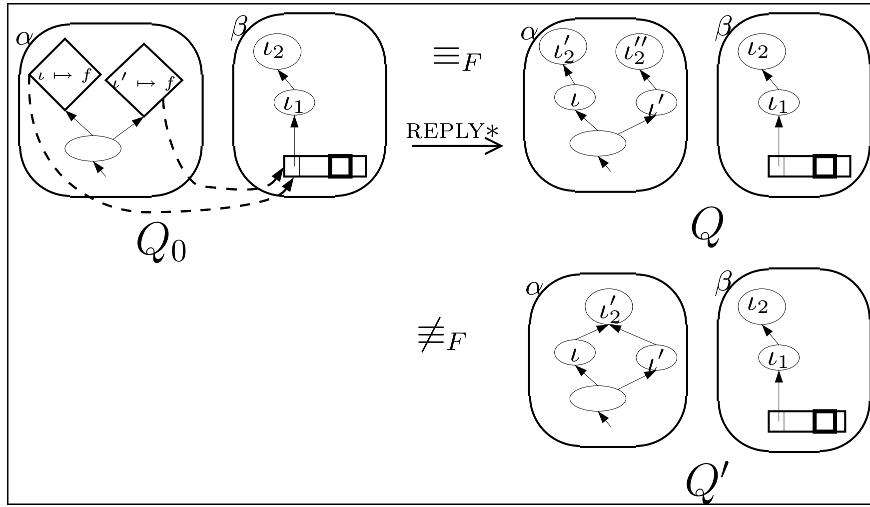


Fig. A.6. Example of alias condition

B

Confluence Proofs

B.1 Context

Property B.1 (Confluence)

$$\begin{cases} P_0 \xrightarrow{*} Q \\ P_0 \xrightarrow{*} Q' \\ Q \bowtie Q' \end{cases} \Longrightarrow Q \curlyvee Q'$$

$$\begin{aligned} \mathcal{Q}(Q, Q') &= \{R | P_0 \xrightarrow{*} R \wedge \forall M \in \mathcal{M}_{\alpha_{P_0}}, \forall \alpha \in R, \\ &\qquad RSL(\alpha_R)\big|_M \trianglelefteq RSL(\alpha_Q)\big|_M \vee RSL(\alpha_R)\big|_M \trianglelefteq RSL(\alpha_{Q'})\big|_M\} \\ &= \{R | P_0 \xrightarrow{*} R \wedge \forall \alpha \in R,\ RSL(\alpha_R)\big|_M \trianglelefteq \big(RSL(\alpha_Q)\big|_M \sqcup RSL(\alpha_{Q'})\big|_M\big)\} \end{aligned}$$

B.2 Lemmas

Lemma B.2 ($\mathcal{Q}$ and compatiblity)

$$\forall R, R' \in \mathcal{Q}(Q, Q'),\ R \bowtie R'$$

Lemma B.3 (Independent stores)

$$dom(\sigma_1) \cap dom(\sigma_2) = \emptyset \Rightarrow \begin{cases} \sigma_1 :: \sigma_2 = \sigma_2 + \sigma_1 \\ \sigma_1 :: \sigma_2 = \sigma_2 :: \sigma_1 \\ \sigma_1 + \sigma_2 = \sigma_2 + \sigma_1 \\ \sigma_1 + (\sigma_2 :: \sigma) = \sigma_2 :: (\sigma_1 + \sigma) \end{cases}$$

Lemma B.4 (Extensibility of local reduction)

$$(a, \sigma) \rightarrow_S (a', \sigma') \Rightarrow (a, \sigma_0 :: \sigma) \rightarrow_S (a'', \sigma_0 :: \sigma'')\ \textit{where}\ (a'', \sigma'') \equiv_F (a', \sigma')$$

Lemma B.5 (*copy* and locations)

$$dom(copy(\iota, \sigma)) \subseteq dom(\sigma)$$

Lemma B.6 (Multiple copies)

$$\iota \in dom(copy(\iota', \sigma')) \;\Rightarrow\; (copy(\iota, \sigma) + copy(\iota', \sigma') = copy(\iota', copy(\iota, \sigma) + \sigma'))$$

Lemma B.7 (Copy and store update)

$$\sigma' + Copy\&Merge(\sigma_1, \iota \;;\; \sigma_2, \iota') \equiv_F Copy\&Merge(\sigma_1, \iota \;;\; \sigma' + \sigma_2, \iota')$$
$$\textit{if} \begin{cases} dom(\sigma') \cap dom(Copy\&Merge(\sigma_1, \iota \;;\; \sigma_2, \iota')) \subseteq dom(\sigma_2) \\ \iota' \notin dom(\sigma') \end{cases}$$

Corollary B.8 (Copy and store update) *If $\iota' \notin dom(\sigma')$ then there is a way of choosing locations allocated by $Copy\&Merge(\sigma_1, \iota \;;\; \sigma_2, \iota')$ such that:*

$$\sigma' + Copy\&Merge(\sigma_1, \iota \;;\; \sigma_2, \iota') \equiv_F Copy\&Merge(\sigma_1, \iota \;;\; \sigma' + \sigma_2, \iota')$$

B.3 Local Confluence

Property B.9 (Diamond property)
Let P be a configuration obtained from P_0: $P_0 \xrightarrow{*} P$:

$$\begin{cases} P \xrightarrow{T_1} P_1 \\ P \xrightarrow{T_2} P_2 \\ P, P_1, P_2 \in \mathcal{Q}(Q, Q') \end{cases} \Longrightarrow P_1 \equiv_F P_2 \vee \exists P_1', P_2', \begin{cases} P_1 \xrightarrow{T_2} P_1' \\ P_2 \xrightarrow{T_1} P_2' \\ P_1' \equiv_F P_2' \\ P_1', P_2' \in \mathcal{Q}(Q, Q') \end{cases}$$

B.4 Calculus with service based on activity name: *Serve*(α)

B.5 Extension

Lemma B.10 ($\equiv_F$ and $\mathcal{Q}(Q, Q')$)
If P is in $\mathcal{Q}$ and P is equivalent modulo future updates to P' then P' is in $\mathcal{Q}$:

$$P \equiv_F P' \wedge P \in \mathcal{Q}(Q, Q') \wedge P_0 \xrightarrow{*} P' \Rightarrow P' \in \mathcal{Q}(Q, Q')$$

Lemma B.11 (REPLY vs. other reduction)

$$P \xrightarrow{\neg\text{REPLY}} R \wedge P \xrightarrow{\text{REPLY}} P' \Rightarrow P' \xrightarrow{\neg\text{REPLY}} R' \wedge R' \equiv_F R$$

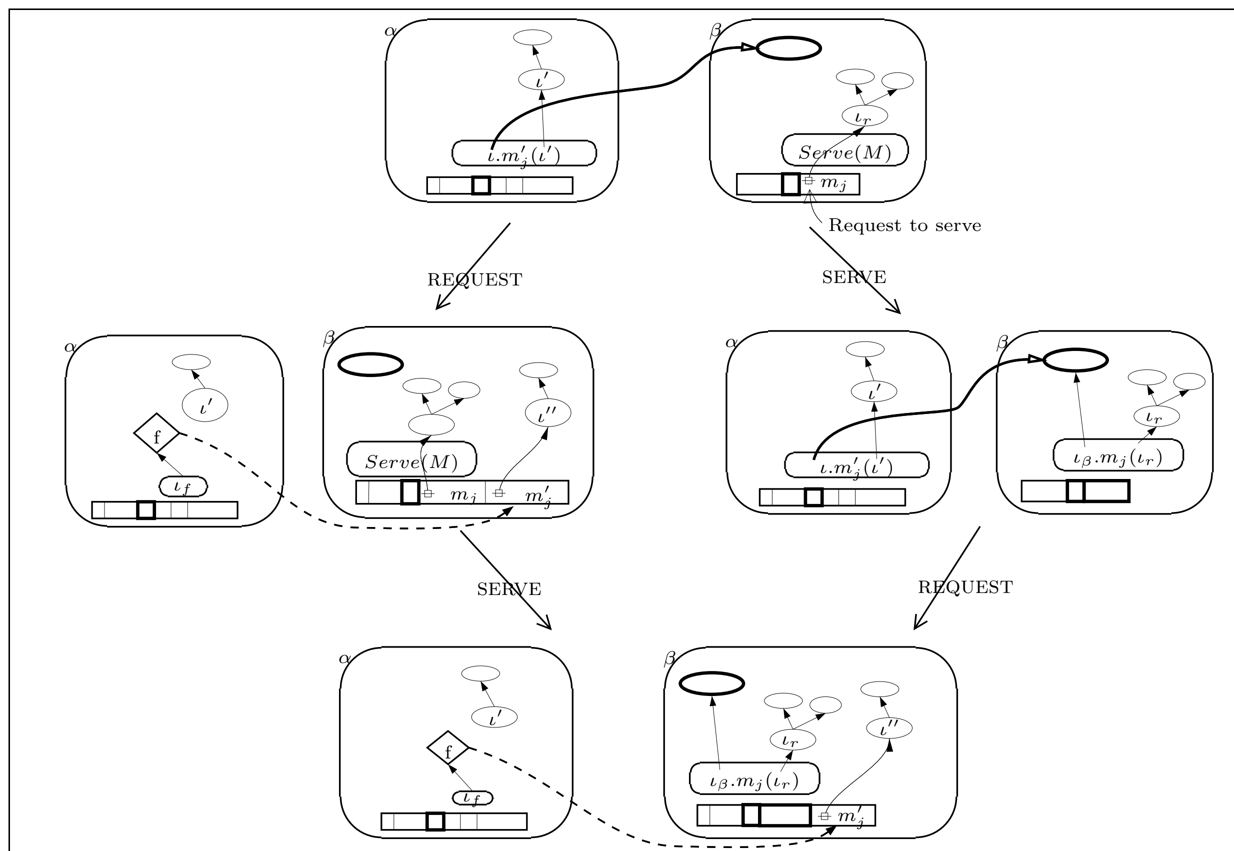


Fig. B.1. SERVE/REQUEST

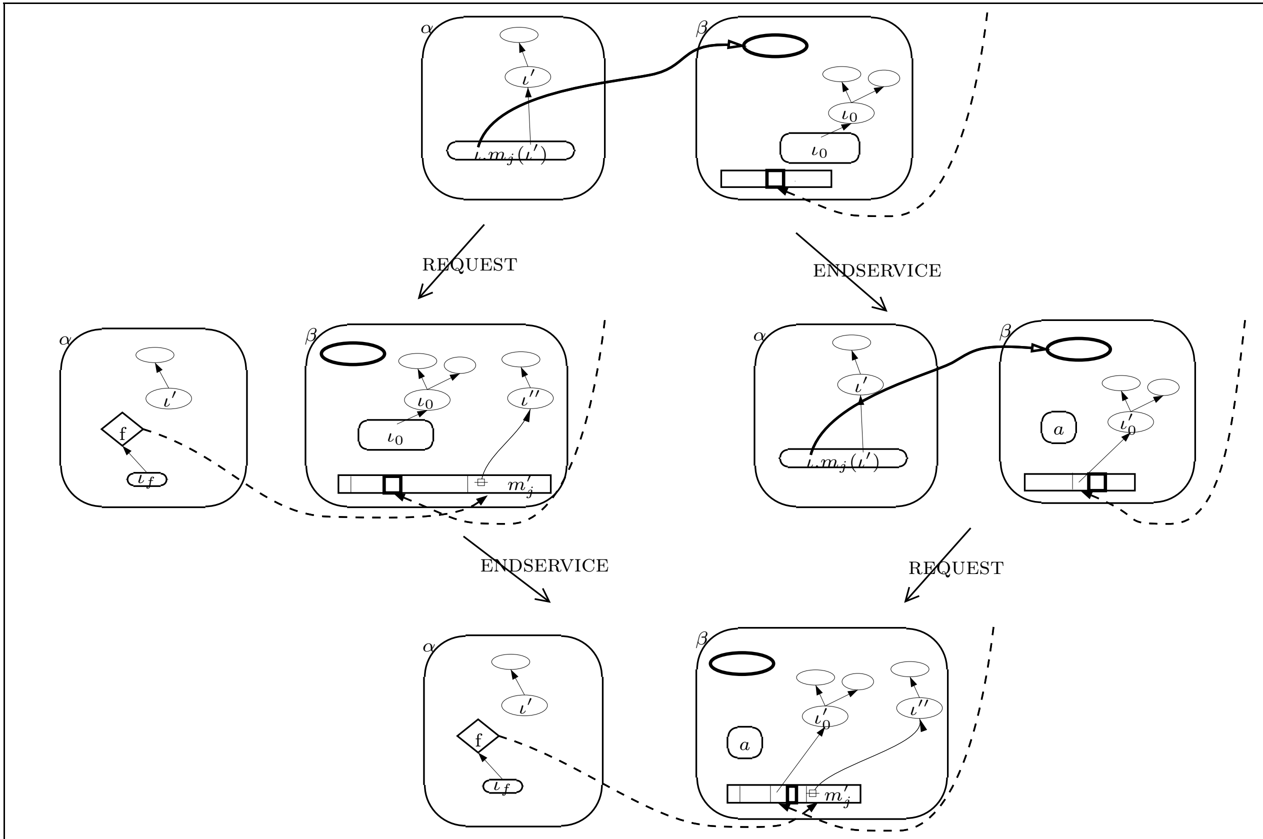


Fig. B.2. ENDSERVICE/REQUEST

Property B.12 (Diamond property with $\equiv_F$)

$$\begin{cases} P_1 \overset{T_1}{\Longrightarrow} Q_1 \\ P_2 \overset{T_2}{\Longrightarrow} Q_2 \\ Q_1, Q_2 \in \mathcal{Q}(Q, Q') \\ P_1 \equiv_F P_2 \end{cases} \Longrightarrow Q_1 \equiv_F Q_2 \vee \exists R_1, R_2, \begin{cases} Q_1 \overset{T_2}{\Longrightarrow} R_1 \\ Q_2 \overset{T_1}{\Longrightarrow} R_2 \\ R_1 \equiv_F R_2 \\ R_1, R_2 \in \mathcal{Q}(Q, Q') \end{cases}$$

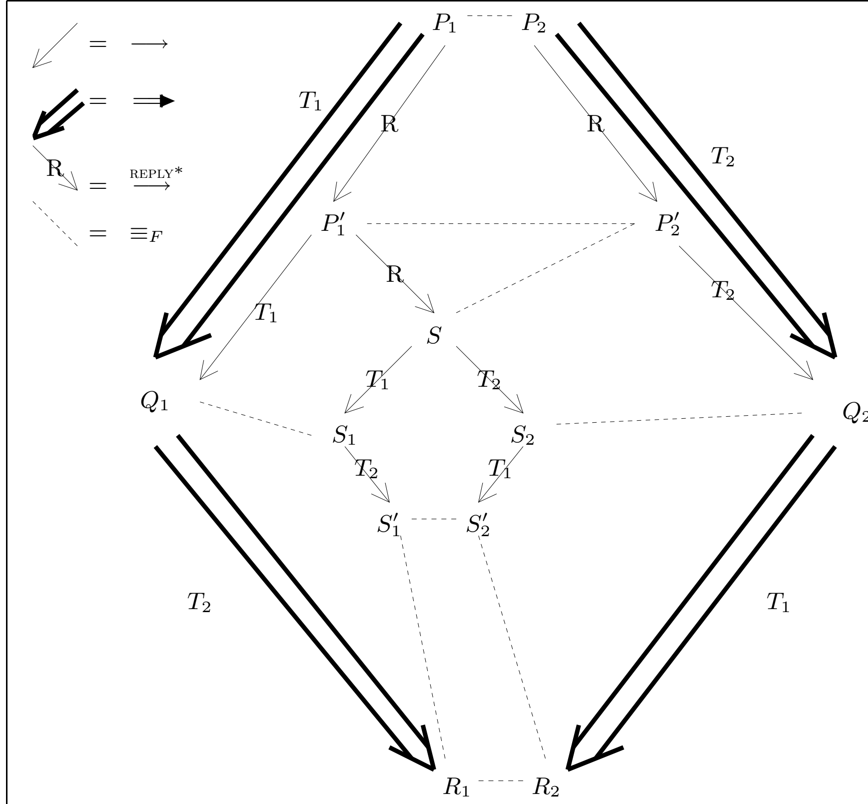


Fig. B.3. The diamond property (Property B.12) proof

References

1. Martín Abadi and Luca Cardelli. An imperative object calculus. In P. D. Mosses, M. Nielsen, and M. I. Schwartzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development*, volume 915 of LNCS, pages 471–485. Springer-Verlag, Berlin, Heidelberg, 1995.
2. Martín Abadi and Luca Cardelli. An imperative object calculus: Basic typing and soundness. In *SIPL '95 – Proceedings of the Second ACM SIGPLAN Workshop on State in Programming Languages*. Technical Report UIUCDCS-R-95-1900, Department of Computer Science, University of Illinois at Urbana–Champaign, 1995.
3. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
4. A Theory of Objects web page. Available at `http://www.luca.demon.co.uk/TheoryOfObjects.html`, and related courses at `http://www.luca.demon.co.uk/TheoryOfObjects/RelatedCourses.html`.
5. Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag, Berlin, Heidelberg, 1997.
6. ACM SIGACT-SIGPLAN. *Conference Record of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95)*, San Francisco, January 22–25, 1995. ACM Press, New York.
7. ACM SIGACT-SIGPLAN. *Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96)*, St. Petersburg, Florida, January 21–24, 1996. ACM Press.
8. Actor foundry. Available at `http://osl.cs.uiuc.edu/foundry/`.
9. Gul Agha. An overview of actor languages. *ACM SIGPLAN Notices*, 21(10):58–67, 1986.
10. Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. Towards a theory of actor computation (extended abstract). In W. R. Cleaveland, editor, *CONCUR'92: Proceedings of the Third International Conference on Concurrency Theory*, pages 565–579. Springer-Verlag, Berlin, Heidelberg, 1992.
11. Gul Agha, Ian A. Mason, Scott F. Smith, and Carolyn L. Talcott. A foundation for actor computation. *Journal of Functional Programming*, 7(1):1–72, 1997.

12. Gul Agha and Prasanna Thati. An algebraic theory of actors and its application to a simple object-based language. In *Essays in Memory of Ole-Johan Dahl*, volume 2635, pages 26–57, 2004.
13. Paulo Sergio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 – Object-Oriented Programming 11th European Conference, Jyvskyl, Finland*, volume 1241, pages 32–59. Springer-Verlag, New York, 1997.
14. Pierre America. Issues in the design of a parallel object-oriented language. *Formal Aspects of Computing*, 1(4):366–411, 1989.
15. Pierre America. Formal techniques for parallel object-oriented languages. In Jos C. M. Baeten and Jan Friso Groote, editors, *CONCUR'91, 2nd International Conference on Concurrency Theory, Amsterdam, The Netherlands, August 26–29, 1991, Proceedings*, volume 527 of LNCS, pages 1–17. Springer-Verlag, Berlin, Heidelberg, 1991.
16. Gregory R. Andrews. *Concurrent programming: principles and practice*. Benjamin-Cummings Publishing Co., Inc., 1991.
17. Arvind, Rishiyur S. Nikhil, and Keshav K. Pingali. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Langanguages and Systems*, 11(4):598–632, 1989.
18. Isabelle Attali, Denis Caromel, and Arnaud Contes. Hierarchical and declarative security for grid applications. In *International Conference on High Performance Computing, HIPC, Hyderabad, India, December 17–20*, LNCS, pages 363–372, Springer-Verlag, Berlin, Heidelberg, 2003.
19. Isabelle Attali, Denis Caromel, and Romain Guider. A step toward automatic distribution of Java programs. In S. F. Smith and C. L. Talcott, editors, *4th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 141–161. Kluwer Academic Publishers, Dordrecht, 2000.
20. Isabelle Attali, Denis Caromel, and Fabrice Huet. Procédé de localisation d'objets mobiles communicants au sein d'un réseau de communications, par transmission d'identifiants de localisation par des répéteurs et mise à jour de serveur, 2003. Patent. Brevet déposé le 23 juillet 2003, No FR 03 08990, copropriété INRIA-Université de Nice – Sophia Antipolis, à l'INPI.
21. Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, flexible, and typed group communications in Java. In *Joint ACM Java Grande – ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press, New York.
22. Henk P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathematics*. North-Holland, Amsterdam, 1981.
23. Françoise Baude, Denis Caromel, Christian Delbé, and Ludovic Henrio. A fault tolerance protocol for ASP calculus: Design and proof. Research Report, INRIA Sophia-Antipolis, June 2004. RR-5246.
24. Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssiere. Communicating mobile active objects in Java. In *Proceedings of HPCN Europe 2000*, volume 1823 of LNCS, pages 633–643. Springer-Verlag, Berlin, Heidelberg, May 2000.
25. Françoise Baude, Denis Caromel, Lionel Mestre, Fabrice Huet, and Julien Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on*

High Performance Distributed Computing, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.
26. Françoise Baude, Denis Caromel, and Matthieu Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3–7 November*, LNCS. Springer Verlag, Berlin, Heidelberg, 2003.
27. Joachim Baumann, Fritz Hohl, and Kurt Rothermel. Mole – concepts of a mobile agent system. Technical Report TR-1997-15, 1997.
28. Nick Benton, Luca Cardelli, and Cédric Fournet. Modern concurrency abstractions for C#. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, pages 415–440. Springer-Verlag, Berlin, Heidelberg, 2002.
29. Gérard Berry. Real time programming: Special purpose or general purpose languages. In G. X. Ritter, editor, *Information Processing 89.* Elsevier Science, Amsterdam, 1989.
30. Gérard Berry and Georges Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
31. Philippe Bidinger and Jean-Bernard Stefani. The kell calculus: operational semantics and type system. In *Proceedings 6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS 03)*, Paris, France, November 2003.
32. Andrew Birrell, Greg Nelson, Susan Owicki, and Edward Wobber. Network objects. *Software – Practice and Experience*, 25(S4):87–130, 1995.
33. R. Blumofe, C. Joerg, B. Kuszmaul, C. Leiserson, K. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th Symposium on Principles and Practice of Parallel Programming*, 1995.
34. Phillip Bogle and Barbara Liskov. Reducing cross-domain call overhead using batched futures. In *Proceedings of OOPSLA '94*, pages 341–359. ACM Press, New York, 1994.
35. Gérard Boudol. Asynchrony and the π-calculus (note). Rapport de Recherche 1702, INRIA Sophia-Antipolis, May 1992.
36. Gérard Boudol. The recursive record semantics of objects revisited. In *Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 269–283. Springer-Verlag, Berlin, Heidelberg, 2001.
37. Rabéa Boulifa and Eric Madelaine. Proof of behaviour properties for proactive programs. Technical Report RR-4460, INRIA, France, 2002. Available at ftp://ftp-sop.inria.fr/pub/rapports/RR-4460.ps.gz.
38. Rabéa Boulifa and Eric Madelaine. Model generation for distributed Java programs. In N. Guelfi, E. Astesiano, and G. Reggio, editors, *Workshop on scientiFic engIneering of Distributed Java applIcations*, volume 2652 of LNCS, Luxembourg, november 2003. Springer-Verlag, Berlin, Heidelberg.
39. Jonathan Bowen's web page on Concurrent Systems. London South Bank University, Centre for Applied Formal Methods. Available at `http://www.jpbowen.com/`, and `http://vl.fmnet.info/concurrent/`.
40. Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

41. Sébastien Briais and Uwe Nestmann. Mobile objects “must” move safely. In *Formal Methods for Open Object-Based Distributed Systems IV – Proceedings of FMOODS'2002, University of Twente, the Netherlands*. Kluwer Academic Publishers, 2002.
42. Kim B. Bruce, Angela Schuett, and Robert van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In Walter G. Olthoff, editor, *Proceedings of ECOOP'95*, volume 952 of LNCS, pages 27–51. Springer-Verlag, Berlin, Heidelberg”, 1995.
43. Roberto Bruni, Losé Meseguer, and Ugo Montanari. Symmetric monoidal and cartesian double categories as a semantic framework for tile logic. *Mathematical Structures in Computer Science*, 12(1):53–90, 2002.
44. A. P. W. Bhm, David C. Cann, R. R. Oldehoeft, and John T. Feo. SISAL reference manual language version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado, 1991.
45. Luca Cardelli. A language with distributed scope. In *Conference Record of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95)* [6], pages 286–297.
46. Luca Cardelli and Andrew D. Gordon. Types for mobile ambients. In *Proceedings of POPL'99*, pages 79–92. ACM Press, New York, 1999.
47. Luca Cardelli and Andrew D. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1):177–213, 2000. An extended abstract appeared in *Proceedings of FoSSaCS '98*, pages 140–155.
48. Denis Caromel. Programming abstractions for concurrent programming. In *Technology of Object-Oriented Languages and Systems,TOOLS Pacific'90*, pages 245–253, 1990.
49. Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, September 1993.
50. Denis Caromel, Christian Delb, Ludovic Henrio, and Romain Quilici. Patent “Dispositif et procédé asynchrones et automatiques de transmission de résultats entre objets communicants”, November 2003. 11/26/2003, No FR 03 138 76.
51. Denis Caromel, Ludovic Henrio, and Bernard Serpette. Asynchronous sequential processes. Research Report, INRIA Sophia-Antipolis, 2003. RR-4753.
52. Denis Caromel, Ludovic Henrio, and Bernard Paul Serpette. Asynchronous and deterministic objects. In *Proceedings of the 31st ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 123–134. ACM Press, 2004.
53. Denis Caromel, Wilfried Klauser, and Julien Vayssière. Towards seamless computing and metacomputing in Java. *Concurrency: Practice and Experience*, 10(11–13):1043–1061, 1998. ProActive available at `http://www.inria.fr/oasis/proactive`.
54. Franck Cassez and Olivier Roux. Compilation of the Electre reactive language into finite transition systems. *Theoretical Computer Science*, 146(1–2):109–143, july 1995.
55. B. Charron-Bost, F. Mattern, and G. Tel. Synchronous, asynchronous, and causally ordered communications. *Distributed Computing*, 9:173–191, September 1996.
56. David G. Clarke, James Noble, and John Potter. Simple ownership types for object containment. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53–76. Springer-Verlag, Berlin, Heidelberg, 2001.

57. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 48–64. ACM Press, New York, 1998.
58. Aramira Corporation. Jumping beans, 1999. `http://www.jumpingbeans.com/`.
59. Pasqua D'Ambra, Marco Danelutto, Daniela di Serafino, and Marco Lapegna. Advanced environments for parallel and distributed applications: a view of current status. *Parallel Comput.*, 28(12):1637–1662, 2002.
60. Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI'94 Conference on Programming Language Design and Implementation*, pages 230–241, Orlando, Florida, June 1994. *ACM SIGPLAN Notices, 29(6)*.
61. Alain Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 226–229, La Jolla, California, June 21–23, 1995.
62. E language. Available at `http://www.erights.org/`.
63. Patrick Thomas Eugster and Sebastien Baehni. Abstracting remote object interaction in a peer-2-peer environment. In *Proceedings of the 2002 joint ACM-ISCOPE Conference on Java Grande*, pages 46–55. ACM Press, New York, 2002.
64. GianLuigi Ferrari and Ugo Montanari. Tiles for concurrent and located calculi. In C. Palamidessi and J. Parrow, editors, *Electronic Notes in Theoretical Computer Science*, volume 7. Elsevier, Amsterdam, 2000.
65. Fabrice Le Fessant. Detecting distributed cycles of garbage in large-scale systems. In *Conference on Principles of Distributed Computing(PODC)*, Rhode Island, August 2001.
66. Cormac Flanagan and Matthias Felleisen. The semantics of future and its use in program optimization. In *Conference Record of the 22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'95)* [6], pages 209–220.
67. Cédric Fournet, Michele Boreale, and Cosimo Laneve. Bisimulations in the join calculus. In *Proceedings of the IFIP Working Conference on Programming Concepts, Methods and Calculi (PROCOMET)*, June 1998.
68. Cédric Fournet and Georges Gonthier. The reflexive CHAM and the join-calculus. In *Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96)* [7], pages 372–385.
69. Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In Larsen et al. [107], pages 844–855.
70. Cédric Fournet, Georges Gonthier, Jean-Jacques Levy, Luc Maranget, and Didier Remy. A calculus of mobile agents. In U. Montanari and V. Sassone, editors, *Proceedings of the 7th International Conference on Concurrency Theory (CONCUR)*, volume 1119 of LNCS, pages 406–421. Springer-Verlag, Berlin, Heidelberg, August 1996.
71. R. J. Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proceedings of PODC'86*, pages 108–120. ACM Press, New York, August 1986.
72. Hubert Garavel, Frédéric Lang, and Radu Mateescu. An overview of CADP 2001. Technical Report RT-254, INRIA, France, 2001. Also in: European

Association for Software Science and Technology (EASST) Newsletter, e 4: 13–24, August 2002.
73. Philippa Gardner's web page and courses. Department of Computing, Imperial College. Available at `http://www.doc.ic.ac.uk/~pg/`, and `http://www.doc.ic.ac.uk/~pg/Concurrency/course.html`.
74. Narain Gehani and William D. Roome. *The concurrent C programming language*. Silicon Press, Summit, New Jersey, 1989.
75. Narain H. Gehani and Thomas A. Cargill. Concurrent programming in the Ada language: The polling bias. *Software – Practice and Experience*, 14(5):413–427, May 1984.
76. Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Enabling primitives for compiling parallel languages. In *Third Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1995.
77. Seth C. Goldstein, Klaus E. Schauser, and David E. Culler. Lazy threads: Implementing a fast parallel call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 1996.
78. Andrew D. Gordon and Paul D. Hankin. A concurrent object calculus: Reduction and typing. In *Proceedings HLCL'98*, volume 16. Elsevier ENTCS, amsterdam, 1998.
79. Andrew D. Gordon, Paul D. Hankin, and Sren B. Lassen. Compilation and equivalence of imperative objects. Research Series RS-97-19, BRICS, Department of Computer Science, University of Aarhus, July 1997. Appears also as Technical Report 429, University of Cambridge Computer Laboratory, June 1997.
80. Andrew D. Gordon, Paul D. Hankin, and Sren B. Lassen. Compilation and equivalence of imperative objects. *FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, 17:74–87, 1997.
81. Andrew D. Gordon and Gareth D. Rees. Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'96)* [7], pages 386–395.
82. James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. Addison-Wesley Longman, Boston, MA, Boston, Massachusets, 2000. Also available at: `http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.htm%l`.
83. Robert H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7(4):501–538, 1985.
84. Kevin Hammond. Parallel functional programming: An introduction (invited paper). In H. Hong, editor, *First International Symposium on Parallel Symbolic Computation (PASCO'94), Linz, Austria*, pages 181–193. World Scientific Publishing, Singapore, 1994.
85. Mark Hapner, Rahul Sharma, Rich Burridge, Joseph Fialli, and Kim Haase. *Java Message Service API tutorial and reference: Messaging for the J2EE platform*. Addison-Wesley Longman, Boston, Massachusets, 2002.
86. Ludovic Henrio. *Asynchronous Object Calculus: Confluence and Determinacy*. PhD thesis, Université de Nice – Sophia-Antipolis – UFR Sciences, November 2003. `http://www.inria.fr/oasis/Ludovic.Henrio/these/`.
87. C. A. R. Hoare. Monitors: An operating system structuring concept. *Communications of the ACM*, 17(10):549–577, October 1974.

88. C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.
89. C. A. R. Hoare. *Communicating sequential processes.* Prentice Hall, Englewood Cliffs, New Jersey, 1985.
90. Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, LNCS, pages 133–147. Springer-Verlag, Berlin, Heidelberg, 1991.
91. Jean D. Ichbiah. Preliminary Ada reference manual. *SIGPLAN Notices*, 14(6a):1–145, 1979.
92. ISO. Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour. ISO/IEC 8807, International Organisation for Standardization, Geneva, Switzerland, 1989.
93. Alan Jeffrey. A distributed object calculus. In *ACM SIGPLAN Workshop Foundations of Object Oriented Languages*, 2000.
94. Cliff Jones. A pi-calculus semantics for an object-based design notation. In Eike Best, editor, *Proceedings of CONCUR'93*, volume 715 of LNCS, pages 158–172. Springer-Verlag, Berlin, Heidelberg, 1993.
95. Cliff B. Jones. An object-based design method for concurrent programs. Technical report, University of Manchester, 1992. UMCS-92-12-1.
96. Cliff B. Jones. Process-algebraic foundations for an object-based design notation. Technical report, University of Manchester, 1993. UMCS-93-10-1.
97. Cliff B. Jones and Steve J. Hodges. Non-interference properties of a concurrent object-based language: Proofs based on an operational semantics. In Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jrg Schek, editors, *Object-Orientation with Parallelism and Persistence*, chapter 1, pages 1–22. Kluwer Academic Publishers, Dordrecht, 1996.
98. Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.
99. Gilles Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475. North-Holland, New York, 1974.
100. Gilles Kahn and David MacQueen. Coroutines and networks of parallel processes. In B. Gilchrist, editor, *Information Processing'77: Proc. IFIP Congress*, pages 993–998. North-Holland, Amsterdam, 1977.
101. Owen Kaser, Shaunak Pawagi, C. R. Ramakrishnan, I. V. Ramakrishnan, and R. C. Sekar. Fast parallel implementation of lazy languages – the EQUALS experience. In *LFP'92: Proceedings of the 1992 ACM conference on LISP and functional programming*, pages 335–344. ACM Press, New York, 1992.
102. Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming*, pages 176–184. ACM Press, New York, 1990.
103. WooYoung Kim and Gul Agha. Efficient support of location transparency in concurrent object-oriented programming languages. In *Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 39. ACM Press, 1995.

104. Naoki Kobayashi, Benjamin C. Pierce, and David N. Turner. Linearity and the pi-calculus. In *Proceedings of POPL '96*, pages 358–371. ACM, January 1996.
105. Naoki Kobayashi and Akinori Yonezawa. Type-theoretic foundations for concurrent object-oriented programing. In *Proceedings of the Ninth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 31–45. ACM Press, New York, 1994.
106. Bernard Lang, Christian Queinnec, and José Piquer. Garbage collecting the world. In *Conference Record of the Nineteenth Annual ACM Symposium on Principles of Programming Languages*, *ACM SIGPLAN Notices*, pages 39–50, January 1992.
107. Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors. *25th Colloquium on Automata, Languages and Programming (ICALP) (Aalborg, Denmark)*, volume 1443 of LNCS. Springer-Verlag, Berlin, Heidelberg, July 1998.
108. Douglas Lea and Doug Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley Longman, Boston, Massachusets, 1996.
109. Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *Communications of the ACM*, 20(8):564–576, August 1977.
110. Xinxin Liu and David Walker. Confluence of processes and systems of objects. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwarzbach, editors, *TAPSOFT '95: Theory and Practice of Software Development, 6th International Joint Conference CAAP/FASE*, volume 915 of LNCS, pages 217–231. Springer-Verlag, Berlin, Heidelberg, 1995.
111. Xinxin Liu and David Walker. Partial confluence of processes and systems of objects. *Theoretical Computer Science*, 206(1–2):127–162, 1998.
112. Olivier Maffes and Axel Poigné. Synchronous automata for reactive, real-time or embedded systems. Technical Report 967, Arbeitspapiere der GMD, Germany, 1996. Available at `http://ais.gmd.de/EES/papers/SAforRRES/SAforRRES.html`.
113. Friedemann Mattern and Stefan Fünfrocken. A non-blocking lightweight implementation of causal order message delivery. In K. P. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems*, volume 938 of LNCS, pages 197–213. Springer-Verlag, Berlin, Heidelberg, 1995.
114. Massimo Merro, Josva Kleist, and Uwe Nestmann. Mobile objects as mobile processes. *Information and Computation*, 177(2):195–241, 2002.
115. Massimo Merro and Davide Sangiorgi. On asynchrony in name-passing calculi. In Larsen et al. [107], pages 856–867.
116. Bertrand Meyer. *Object-oriented software construction (2nd edition)*. Prentice-Hall, Englewood Cliffs, New Jersey, 1997.
117. Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersey, 1989. SU Fisher Research 511/24.
118. Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of Series F. NATO ASI, Springer-Verlag, Berlin, Heidelberg, 1993. Available as Technical Report ECS-LFCS-91-180, University of Edinburgh, October 1991.
119. Robin Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, May 1999.

120. Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, part I/II. *Journal of Information and Computation*, 100:1–77, September 1992.
121. D. S. Milojičić, W. LaForge, and D. Chauhan. Mobile Objects and Agents (MOA). In *Proceedings of USENIX COOTS '98, Santa Fe, New Mexico*, April 1998.
122. Uwe Nestmann, Hans Huttel, Josva Kleist, and Massimo Merro. Aliasing models for object migration. In *European Conference on Parallel Processing*, pages 1353–1368, 1999.
123. Uwe Nestmann, Hans Hüttel, Josva Kleist, and Massimo Merro. Aliasing models for mobile objects. *Information and Computation*, 175(1):3–33, 2002.
124. Uwe Nestmann and Martin Steffen. Typing confluence. In Stefania Gnesi and Diego Latella, editors, *Proceedings of FMICS'97*, pages 77–101. Consiglio Nazionale Ricerche di Pisa, 1997. Also available as report ERCIM-10/97-R052, European Research Consortium for Informatics and Mathematics, 1997.
125. Uwe Nestmann's web page on Calculi for Mobile Processes. Ecole Polytechnique Fédérale de Lausanne (EPFL). Available at `http://lamp.epfl.ch/~uwe/`, and `http://lamp.epfl.ch/mobility/`.
126. Bradford Nichols, Bick Buttlar, and Jackie Proulx Farrell. *Pthreads Programming*. O'Reilly & Associates, Inc., 981 Chestnut Street, Newton, MA 02164, USA, 1996.
127. Martin Odersky. Functional nets. In Gert Smolka, editor, *Proceedings of ESOP 2000*, volume 1782 of LNCS, pages 1–25. Springer-Verlag, Berlin, Heidelberg, 2000.
128. Olaf Owe, Stein Krogdahl, and Tom Lyche, editors. *From Object-Orientation to Formal Methods, Essays in Memory of Ole-Johan Dahl*, volume 2635 of LNCS. Springer, 2004.
129. Thomas Parks and David Roberts. Distributed Process Networks in Java. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS2003)*, Nice, France, April 2003.
130. Benjamin C. Pierce. Programming in the pi-calculus: A tutorial introduction to Pict (Pict version 4.1), 1998.
131. Benjamin C. Pierce and David N. Turner. Concurrent objects in a process calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Proceedings of Theory and Practice of Parallel Programming (TPPP'94), Sendai, Japan*, LNCS, pages 187–215. Springer-Verlag, Berlin, Heidelberg, 1995.
132. Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, Cambridge, Massachusets, May 2000.
133. *POSIX. System Application Program Interface (API) [C Language]*. Information technology – Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1996. ISO/IEC 9945-1:1996.
134. ProActive API and environment. Available at `http://www.inria.fr/oasis/proactive` (under LGPL).
135. Christian Queinnec and Kathleen Callaway. *Lisp in small pieces*. Cambridge University Press, 1996, reprint 2003.

136. John H. Reppy. CML: A higher concurrent language. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 293–305. ACM Press, New York, 1991.
137. Rémi Revire, Florence Zaral, and Thierry Gautier. Efficient and easy parallel implementation of large numerical simulations. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, volume 2840 of LNCS, pages 663–666. Springer-Verlag, Berlin, Heidelberg, October 2003.
138. Jean-Louis Roch. Ordonnancement de programmes parallèles sur grappes : théorie versus pratique. In *Actes du Congrès International ALA 2001, Universit Mohamm V*, pages 131–144, Rabat, Maroc, 28–31 May 2001.
139. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Lecture Notes in Computer Science*, 915:651–??, 1995.
140. Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science*, 167(1–2):131–170, 1996.
141. Davide Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).
142. Davide Sangiorgi. The typed π-calculus at work: A proof of Jones's parallelisation theorem on concurrent objects. *Theory and Practice of Object-Oriented Systems*, 5(1), 1999. An early version was included in the *Informal Proceedings of FOOL 4*, January 1997.
143. Davide Sangiorgi. Asynchronous process calculi: The first-order and higher-order paradigms (tutorial). *Theoretical Computer Science*, 253(2):311–350, February 2001.
144. Davide Sangiorgi and David Walker. *The π-calculus: a Theory of Mobile Processes*. Cambridge University Press, 2001.
145. Alan Schmitt and Jean-Bernard Stefani. The M-calculus: A higher-order distributed process calculus. In *Proceedings of the 30th ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, pages 50–61. ACM Press, New York, 2003.
146. Marc Shapiro, Peter Dickman, and David Plainfoss. SSP chains: Robust, distributed references supporting acyclic garbage collection. Rapport de Recherche 1799, INRIA, Rocquencourt, November 1992.
147. S. Skedzielewski and J. Glauert. *IF1 An Intermediate Form for Applicative Languages*. Lawrence Livermore National Laboratory, Livermore, California, USA, 1985.
148. David B. Skillicorn and Domenico Talia. *Programming languages for parallel processing*. IEEE Computer Society Press, 1995.
149. David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, 1998.
150. Brian Cantwell Smith. Reflection and semantics in Lisp. In *Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, Salt Lake City, Utah, January 15–18, 1984. ACM Press, New York.
151. Guy L. Steele, Jr. Making asynchronzous parallelism safe for the world. In *POPL'90. Proceedings of the Seventeenth Annual ACM Symposium on Prin-*

ciples of Programming Languages, January 17–19, 1990, San Francisco, CA, pages 218–231. ACM Press, New York, 1990.

152. Jean Bernard Stefani. A calculus of higher-order distributed components. Technical report, INRIA Rhones Alpes, 2003. RR-4692.
153. Chris Steketee, Weiping Zhu, and Philip Moseley. Implementation of process migration in amoeba. In *International Conference on Distributed Computing Systems*, pages 194–201, 1994.
154. Chantal Taconet and Guy Bernard. A localization service for large scale distributed systems based on microkernel technology. In *Proceedings of ROSE'94*, 1994.
155. Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Englewood Cliffs, New Jersey, 2002.
156. Éric Tanter, Jacques Noyé, Denis Caromel, and Pierre Cointe. Partial behavioral reflection: Spatial and temporal selection of reification. In Ron Crocker and Guy L. Steele, Jr., editors, *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented Programing, Systems, Languages, and Applications (OOPSLA 2003)*, pages 27–46, Anaheim, California, October 2003. ACM Press, New York.
157. P. H. J. van Eijk, C. A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS – Results of the ESPRIT/SEDOS Project*. North-Holland, Amsterdam, 1989.
158. Takuo Watanabe and Akinori Yonezawa. Reflection in an object-oriented concurrent language. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 306–315. ACM Press, New York, 1988.
159. Daren L. Webb, Andrew L. Wendelborn, and Julien Vayssière. A study of computational reconfiguration in a Process Network. In *Proceedings of the 7th Workshop on Integrated Data Environments Australia (IDEA'7)*, February 2000.
160. Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990. Expansion of OOPSLA'89 Keynote Talk, October 4.
161. Akinori Yonezawa, Jean-Pierre Briot, and Etsuya Shibayama. Object-oriented concurrent programming in ABCL/1. In *Proceedings OOPSLA'86*, pages 258–268, November 1986. Published as *ACM SIGPLAN Notices*, 21.
162. Akinori Yonezawa, Etsuya Shibayama, Toshihiro Takada, and Yasuaki Honda. Modelling and programming in an object-oriented concurrent language ABCL/1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 55–89. MIT Press, Cambridge, Massachusets, 1987.
163. Silvano Dal Zilio. Mobile processes: A commented bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000, Nantes, France, June 19–23, 2000*. Springer-Verlag, Berlin, Heidelberg, 2001.

Notation

Concepts

Term	Definition	Page
Active object	Root object of an activity	
Activity	A process made of a single active object and a set of passive objects	
Wait-by-necessity	Blocking of execution upon a strict operation on a future: $\alpha[\mathcal{R}[\iota\ldots],\sigma_\alpha\ldots] \wedge \sigma_\alpha(\iota) = fut(f_i^{\gamma\to\beta})$	
Service method	Method started upon activation: m_j in $Active(a, m_j)$	36
Request	Asynchronous remote method call	
Future	Represents the result of a request before the response is sent back	
Future value	Value associated to a future $f_i^{\alpha\to\beta}$ $copy(\iota,\sigma)$ where $\{f_i^{\alpha\to\beta} \mapsto \iota\} \in F_\alpha$	
Computed future	A future which has a value associated: $f_i^{\alpha\to\beta}$ where $f_i^{\alpha\to\beta} \in dom(F_\beta)$	
Not updated future	A computed future not yet locally updated	
Partial future value	Future value containing references to futures	
Closed term	Term without free variable ($fv(a) = \emptyset$)	
Source term	Closed term without location: $fv(a) = \emptyset \wedge locs(a) = \emptyset$	
Reduced object	Object with all fields reduced to a location: $o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]_{j\in 1..m}^{i\in 1..n}$	47
Potential services	Static approximation of the set of method names appearing in service primitives $\mathcal{M}_{\alpha_P}$	176
Interfering requests	Two requests that can be served by the same serve primitive: $[m_1;\iota;f_i^{\alpha\to\beta}]$ and $[m_2;\iota;f_i^{\gamma\to\beta}]$ such that there is $M, \{m_1, m_2\} \subseteq M$ and $Serve(M)$ can appear in β	

Cycle of futures	set of future identifiers $\{fut(f_i^{\gamma_i\to\beta_n})\}$ such that $\beta_0\ldots\beta_n$ verify: $\begin{cases}\forall i,\, 0<i\le n, fut(f_{i-1}^{\gamma_{i-1}\to\beta_{i-1}})\in copy(F_{\beta_i}(fut(f_i^{\gamma_i\to\beta_i})),\sigma_{\beta_i})\\ fut(f_n^{\gamma_n\to\beta_n})\in copy(F_{\beta_0}(fut(f_0^{\gamma_0\to\beta_0})),\sigma_{\beta_0})\end{cases}$	63

Core Syntax: ASP Source Terms

$[l_i = b_i;\ m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$	Object definition	31
$a.l_i$	Field access	31
$a.l_i := b$	Field update	31
$a.m_j(b)$	Method call	31
$clone(a)$	Superficial copy	31
$Active(a, m_j)$	Object activation	36
$Serve(M)$	Request service	36
M	list of method labels	36

Encoding of Classical Primitives

$let\ x = a\ in\ b$	$[; m = \varsigma(z, x)b].m(a)$	31
$a; b$	$[; m = \varsigma(z, z')b].m(a)$	31
$\lambda x.b$	$[arg = [], val = \varsigma(x)b\{\!\{x \leftarrow x.arg\}\!\}]$	31
$(b\ a)$	$(clone(b).arg := a).val()$	31
$Repeat(a)$	$[; repeat = \varsigma(x)a; x.repeat()].repeat()$	48
$FifoService$	$Repeat(Serve(\mathcal{M}))$	48
$Repeat\ a\ Until\ b$	$[; rep = \varsigma(x)a;\ if\ (not(b))\ then\ x.rep()].rep()$	48

ASP Intermediate Terms and Semantic Structures

(a, σ)	Sequential configuration	
α, β	Activities: $\alpha[a_\alpha; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha]$	35
	[current term, store, active object, future values, pending requests, current future]	47
ι	Locations	31
$locs(a)$	Set of locations occurring in a	
P, Q	Parallel configuration	47
$\alpha \in P$	Activity α belongs to the configuration P	
$a \Uparrow f, b$	a with continuation b	36
	f is the future associated with the continuation	
$AO(\alpha)$	Generalized reference to the activity α	47
$f_i^{\alpha\to\beta}$	Future identifier	
$fut(f_i^{\alpha\to\beta})$	Future reference	47

$r = [m_j; \iota; f_i^{\alpha \to \beta}]$	Request: asynchronous remote method call	47
$R_\alpha = \{[m_j; \iota; f_i^{\alpha \to \beta}]\}$	Pending requests: a queue of requests	47
$R :: r$	Adds a request r at the end of the pending requests R	
$r :: R$	Takes the first request r at the beginning of the pending requests	
$F :: \{f_i \mapsto \iota\}$	Adds a new future association to the future values	47

General Notation

$\{a_i\}$	List	32
$\{a \mapsto b\}$	Association/finite mapping	32
$\theta ::= \{\!\!\{b \leftarrow c\}\!\!\}$	Substitution	32
$\xrightarrow{*}$	Transitive closure of any reduction $\to$	176
$\bigoplus$	Disjoint union	56
$L\big\vert_M$	Restriction of (RSL) list L to labels belonging to M	59
L_n	n^{th} element of the list L	59
$\sqcup$	Least upper bound	59
$\exists^1$	There is at most one	67

Stores

σ	Store: finite map from locations to objects (reduced or generalized reference) $\sigma ::= \{\iota_i \mapsto o_i\}$	32
$dom(\sigma)$	set of locations defined by σ	
$\sigma :: \sigma'$	Append of disjoint stores	
$\sigma + \sigma'$	Updates the values defined in σ' by those defined in σ: $(\sigma + \sigma')(\iota) = \begin{cases} \sigma(\iota) & \text{if } \iota \in dom(\sigma) \\ \sigma'(\iota) & \text{otherwise} \end{cases}$	32
$Merge(\iota, \sigma, \sigma')$	Store merge: merges σ and σ' independently except for ι which is taken from σ': $Merge(\iota, \sigma, \sigma') = \sigma'\theta + \sigma$ where $\theta = \{\!\!\{\iota' \leftarrow \iota'' \mid \iota' \in dom(\sigma') \cap dom(\sigma) \backslash \{\iota\}, \iota'' \text{ fresh}\}\!\!\}$	48
$copy(\iota, \sigma)$	Deep copy of $\sigma(\iota)$	48
$Copy\&Merge(\sigma, \iota\ ;\ \sigma', \iota')$	Appends in $\sigma'(\iota')$ a deep copy of $\sigma(\iota)$ $= Merge(\iota', \sigma', copy(\iota, \sigma)\{\!\!\{\iota \leftarrow \iota'\}\!\!\})$	48

Semantics

$\mathcal{R}$	Reduction context	32,48
$\mathcal{R}[a]$	Substitution inside a reduction context	32
$\rightarrow_S$	Sequential reduction	33
$\longrightarrow$	Parallel reduction	50
$\xrightarrow{T}$	Parallel reduction where rule T is applied	
$\Longrightarrow$	Parallel reduction with future updates, i.e., Parallel reduction preceded by some reply rules	63 150
$\stackrel{T}{\Longrightarrow}$	Parallel Reduction with future updates where rule T is applied: $\xrightarrow{\text{REPLY}*}\xrightarrow{T}$ if $T \neq \text{REPLY}$ and $\xrightarrow{\text{REPLY}*}$ if $T = \text{REPLY}$	63
$FL(\alpha)$	Future List of α	54
$RSL(\alpha)$	Request Sender List of α: $(RSL(\alpha))_n = \beta^f$ if $f_n^{\beta\rightarrow\alpha} \in FL(\alpha)$	59
$\trianglelefteq$	RSL comparison: prefix order on sender activities	59
$\mathcal{M}_{\alpha_P}$	Potential services: Static approximation of the set of M that can appear in the $Serve(M)$ instructions of α_P: $P \xrightarrow{*} Q \wedge Q = \alpha[\mathcal{R}[Serve(M)], \ldots] \parallel \ldots$ $\Rightarrow \exists M' \in \mathcal{M}_{\alpha_P},\, M \subseteq M'$	
$ActiveRefs(\alpha)$	Set of active objects referenced by α: $\{\beta \mid \exists\iota \in dom(\sigma_\alpha),\, \sigma_\alpha(\iota) = AO(\beta)\}$	51
$FutureRefs(\alpha)$	Set of futures referenced by α: $\{f_i^{\beta\rightarrow\gamma} \mid \exists\iota \in dom(\sigma_\alpha),\, \sigma_\alpha(\iota) = fut(f_i^{\beta\rightarrow\gamma})\}$	51
FF	Set of Forwarded Futures: $\{(f_i^{\alpha\rightarrow\beta}, \gamma, \delta)\} \in FF$ if $f_i^{\alpha\rightarrow\beta}$ has been transmitted from γ to δ	117

Equivalences

$\equiv$	Equality modulo renaming (alpha conversion) of locations and futures, and reordering of pending requests	
$\equiv_F$	Equivalence modulo future updates also called equivalence modulo replies	61

Properties

$\vdash P$ OK	Well-formed configuration	54
$RSL(\alpha) \bowtie RSL(\beta)$	RSL compatibility	59
$P \bowtie Q$	Configuration compatibility	59
$P_1 \curlyvee P_2$	Configuration confluence: $\exists R_1, R_2, P_1 \xrightarrow{*} R_1 \wedge P_2 \xrightarrow{*} R_2 \wedge R_1 \equiv_F R_2$	64
$\mathcal{G}(P_0)$	Approximated call graph α can send a request foo to β implies $(\dot{\alpha}, \dot{\beta}, foo) \in \mathcal{G}(P_0)$	67
Request flow graph	$\alpha \rightarrow_R \beta$ if α has sent a request to β	69
$DON(P)$	Deterministic Object Network	67
$SDON(P)$	Static Deterministic Object Network	68
$TDON(P)$	Tree Deterministic Object Network	69

Syntax of ASP Calculus

Source terms

$a, b \in L ::= x$	variable	
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$	object definition	
$\mid a.l_i$	field access	
$\mid a.l_i := b$	field update	
$\mid a.m_j(b)$	method call	
$\mid clone(a)$	superficial copy	where
$\mid Active(a, m_j)$	activates object: deep copy + activity creation m_j is the activity method or $\emptyset$ for FIFO service	
$\mid Serve(M)$	Serves a request among a set of method labels	

M is a set of method labels used to specify which request has to be served.

$$M = m_1, \ldots, m_k$$

Intermediate Terms

Terms

$a, b \in L' ::= x$	variable
$\mid [l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	object definition
$\mid a.l_i$	field access
$\mid a.l_i := b$	field update
$\mid a.m_j(b)$	method call
$\mid clone(a)$	superficial copy
$\mid Active(a, m_j)$	object activation
$\mid Serve(M)$	service primitive
$\mid \iota$	location
$\mid a \Uparrow f, b$	a with continuation b

Configurations

$$P, Q ::= \alpha[a; \sigma; \iota; F; R; f] \parallel \beta[\ldots] \parallel \ldots$$

Requests

$$R ::= \{[m_j; \iota; f_i^{\alpha \to \beta}]\}$$

Future Values

$$F ::= \{f_i^{\gamma \to \alpha} \mapsto \iota\}$$

Store

$$\sigma ::= \{\iota_i \mapsto o_i\}$$

$o ::= [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m}$	reduced object
$\mid AO(\alpha)$	active object reference
$\mid fut(f_i^{\alpha \to \beta})$	future reference

Operational Semantics

STOREALLOC:

$$\frac{\iota \notin dom(\sigma)}{(\mathcal{R}[o], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \mapsto o\} :: \sigma)}$$

FIELD:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n}{(\mathcal{R}[\iota.l_k], \sigma) \rightarrow_S (\mathcal{R}[\iota_k], \sigma)}$$

INVOKE:

$$\frac{\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..m}{(\mathcal{R}[\iota.m_k(\iota')], \sigma) \rightarrow_S (\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}], \sigma)}$$

UPDATE:

$$\frac{\begin{array}{c}\sigma(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..n}_{j \in 1..m} \quad k \in 1..n \\ o' = [l_i = \iota_i; l_k = \iota'; l_{k'} = \iota_{k'}; m_j = \varsigma(x_j, y_j)a_j]^{i \in 1..k-1,\, k' \in k+1...n}_{j \in 1..m}\end{array}}{(\mathcal{R}[\iota.l_k := \iota'], \sigma) \rightarrow_S (\mathcal{R}[\iota], \{\iota \rightarrow o'\} + \sigma)}$$

CLONE:

$$\frac{\iota' \notin dom(\sigma)}{(\mathcal{R}[clone(\iota)], \sigma) \rightarrow_S (\mathcal{R}[\iota'], \{\iota' \mapsto \sigma(\iota)\} :: \sigma)}$$

Table 1. Sequential reduction

$$\iota \in dom(copy(\iota,\sigma))$$
$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow locs(\sigma(\iota')) \subseteq dom(copy(\iota,\sigma))$$
$$\iota' \in dom(copy(\iota,\sigma)) \Rightarrow copy(\iota,\sigma)(\iota') = \sigma(\iota')$$

Table 2. Deep copy

LOCAL:
$$\frac{(a,\sigma) \rightarrow_S (a',\sigma') \quad \rightarrow_S \text{does not clone a future}}{\alpha[a;\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a';\sigma';\iota;F;R;f] \parallel P}$$

NEWACT:
$$\frac{\begin{array}{c}\gamma \text{ fresh activity} \quad \iota' \notin dom(\sigma) \quad \sigma' = \{\iota' \mapsto AO(\gamma)\} :: \sigma \\ \sigma_\gamma = copy(\iota'',\sigma) \quad Service = (\text{if } m_j = \emptyset \text{ then } FifoService \text{ else } \iota''.m_j())\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[Active(\iota'',m_j)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota'];\sigma';\iota;F;R;f] \parallel \gamma[Service;\sigma_\gamma;\iota'';\emptyset;\emptyset;\emptyset] \parallel P\end{array}}$$

REQUEST:
$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \quad f_i^{\alpha\rightarrow\beta} \text{ new future} \quad \iota_f \notin dom(\sigma_\alpha) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha,\iota' \; ; \; \sigma_\beta,\iota'') \quad \sigma'_\alpha = \{\iota_f \mapsto fut(f_i^{\alpha\rightarrow\beta})\} :: \sigma_\alpha\end{array}}{\begin{array}{c}\alpha[\mathcal{R}[\iota.m_j(\iota')];\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[\mathcal{R}[\iota_f];\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma'_\beta;\iota_\beta;F_\beta;R_\beta :: [m_j;\iota'';f_i^{\alpha\rightarrow\beta}];f_\beta] \parallel P\end{array}}$$

SERVE:
$$\frac{R = R' :: [m_j;\iota_r;f'] :: R'' \quad m_j \in M \quad \forall m \in M, m \notin R'}{\begin{array}{c}\alpha[\mathcal{R}[Serve(M)];\sigma;\iota;F;R;f] \parallel P \longrightarrow \\ \alpha[\iota.m_j(\iota_r) \Uparrow f, \mathcal{R}[[]];\sigma;\iota;F;R' :: R'';f'] \parallel P\end{array}}$$

ENDSERVICE:
$$\frac{\iota' \notin dom(\sigma) \quad F' = F :: \{f \mapsto \iota'\} \quad \sigma' = Copy\&Merge(\sigma,\iota \; ; \; \sigma,\iota')}{\alpha[\iota \Uparrow (f',a);\sigma;\iota;F;R;f] \parallel P \longrightarrow \alpha[a;\sigma';\iota;F';R;f'] \parallel P}$$

REPLY:
$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\rightarrow\beta}) \quad F_\beta(f_i^{\gamma\rightarrow\beta}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta,\iota_f \; ; \; \sigma_\alpha,\iota)}{\begin{array}{c}\alpha[a_\alpha;\sigma_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha;\sigma'_\alpha;\iota_\alpha;F_\alpha;R_\alpha;f_\alpha] \parallel \beta[a_\beta;\sigma_\beta;\iota_\beta;F_\beta;R_\beta;f_\beta] \parallel P\end{array}}$$

Table 3. Parallel reduction (used or modified values are non-gray)

Overview of Properties

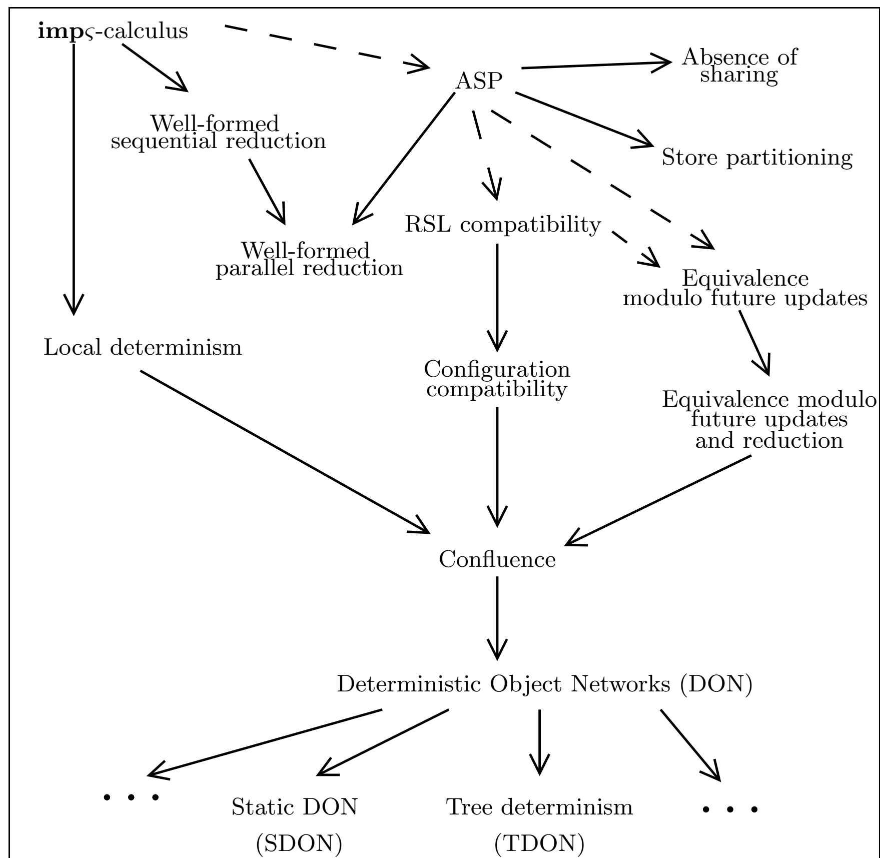


Fig. 2. Diagram of properties

Overview of ASP Extensions

We present here most of the features that have been added to ASP in Part IV. We provide a brief summary, based on the syntax, and most of the reduction rules associated with these features. When several and somehow equivalent reduction rules exist for the same feature, we choose one of them.

Three Confluent Features:

1. Delegation

Delegates to another activity the responsibility to reply to the current request (confluent).

Syntax

$$delegate(a)$$

Reduction Rules

Parallel DELEGATE:

$$\frac{\begin{array}{c}\sigma_\alpha(\iota) = AO(\beta) \quad \iota'' \notin dom(\sigma_\beta) \\ \sigma'_\beta = Copy\&Merge(\sigma_\alpha, \iota' \; ; \; \sigma_\beta, \iota'') \quad f_\emptyset \text{ new future}\end{array}}{\begin{array}{l}\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_i^{\gamma\to\alpha}] \parallel \beta[a_\beta; \sigma_\beta; \iota_\beta; F_\beta; R_\beta; f_\beta] \parallel P \longrightarrow \\ \quad \alpha[\mathcal{R}[[]]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\emptyset] \parallel \beta[a_\beta; \sigma'_\beta; \iota_\beta; F_\beta; R_\beta :: [m_j; \iota''; f_i^{\gamma\to\alpha}]; f_\beta] \parallel P\end{array}}$$

Sequential DELEGATE:

$$\frac{\sigma_\alpha(\iota) = [l_i = \iota_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m} \quad k \in 1..m}{\begin{array}{l}\alpha[\mathcal{R}[delegate(\iota.m_j(\iota'))]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P \longrightarrow \\ \quad \alpha[\mathcal{R}[a_k\{\!\{x_k \leftarrow \iota, y_k \leftarrow \iota'\}\!\}]; \sigma_\alpha; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel P\end{array}}$$

Generalized REPLY:

$$\frac{\sigma_\alpha(\iota) = fut(f_i^{\gamma\to\delta}) \quad F_\beta(f_i^{\gamma\to\delta}) = \iota_f \quad \sigma'_\alpha = Copy\&Merge(\sigma_\beta, \iota_f\ ;\ \sigma_\alpha, \iota)}{\begin{array}{c}\alpha[a_\alpha; \boldsymbol{\sigma_\alpha}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \boldsymbol{\sigma_\beta}; \iota_\beta; \boldsymbol{F_\beta}; R_\beta; f_\beta] \parallel P \longrightarrow \\ \alpha[a_\alpha; \boldsymbol{\sigma'_\alpha}; \iota_\alpha; F_\alpha; R_\alpha; f_\alpha] \parallel \beta[a_\beta; \boldsymbol{\sigma_\beta}; \iota_\beta; \boldsymbol{F_\beta}; R_\beta; f_\beta] \parallel P\end{array}}$$

2. Explicit Wait

Waits for a future update (confluent).

Syntax

$$waitFor(a)$$

Encoding

$$[\![[l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}]\!] \triangleq [wait = [], l_i = b_i; m_j = \varsigma(x_j, y_j)a_j]^{i\in 1..n}_{j\in 1..m}$$
$$[\![waitFor(a)]\!] \triangleq a.wait$$

3. Method Update

Changes the code associated to a method (confluent).

Syntax

$$x.foo \Leftarrow b$$

Five Non-confluent Features:

1. Testing Future Reception

Returns "true" if a future is awaited, and "false" if it has already been updated.

Syntax

$$awaited(a)$$

Reduction Rules

WAITT:

$$\frac{\sigma(\iota) = fut(f_i^{\alpha\rightarrow\beta})}{(\mathcal{R}[awaited(\iota)], \sigma) \rightarrow_S (\mathcal{R}[true], \sigma)}$$

WAITF:

$$\frac{\sigma(\iota) \neq fut(f_i^{\alpha\rightarrow\beta})}{(\mathcal{R}[awaited(\iota)], \sigma) \rightarrow_S (\mathcal{R}[false], \sigma)}$$

2. Non-blocking Service

Serves a request if it is in the request queue, else continues the execution.

Syntax

$$ServeWithoutBlocking(M)$$

Reduction Rules

SERVEWBSERVE:

$$\frac{R = R' :: [m_j; \iota_r; f'] :: R'' \quad m_j \in M \quad \forall m \in M, m \notin R'}{\begin{array}{c}\alpha[\mathcal{R}[ServeWithoutBlocking(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \\ \alpha[\iota.m_j(\iota_r) \Uparrow f, \mathcal{R}[[]]; \sigma; \iota; F; R' :: R''; f'] \parallel P\end{array}}$$

SERVEWBCONTINUE:

$$\frac{\forall m \in M, m \notin R}{\alpha[\mathcal{R}[ServeWithoutBlocking(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[[]]; \sigma; \iota; F; R; f] \parallel P}$$

3. Testing Request Reception

Returns “true” if a corresponding request is in the request queue.

Syntax

$$inQueue(M)$$

Reduction Rules

INQUEUET:

$$\frac{\exists m \in M,\ m \in R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[true]; \sigma; \iota; F; R; f] \parallel P}$$

INQUEUEF:

$$\frac{\forall m \in M,\ m \notin R}{\alpha[\mathcal{R}[inQueue(M)]; \sigma; \iota; F; R; f] \parallel P \longrightarrow \alpha[\mathcal{R}[false]; \sigma; \iota; F; R; f] \parallel P}$$

4. Join Pattern Example

The term below encodes a join pattern cell: the cell reacts to the simultaneous presence of two messages, either s and set, or s and get. s is used to store the internal state of the cell.

Encoding a Join Pattern Cell

$$\begin{aligned}
Cell \triangleq Active([&s_v = [],\ set_v = [];\\
&set = \varsigma(this, v)this.set_v := v\\
&s = \varsigma(this, v)this.s_v := v\\
&get = \varsigma(this)[]\\
&srv = \varsigma(this)Repeat(if\ inQueue(s) \wedge inQueue(set)\ then\\
&\qquad\qquad\qquad\qquad\qquad this.setcell()\\
&\qquad\qquad\qquad if\ inQueue(s) \wedge inQueue(get)\ then\\
&\qquad\qquad\qquad\qquad\qquad this.getcell()),\\
&setcell() = \varsigma(this)(Serve(set); Serve(s); thisActivity.s(set_v)),\\
&getcell() = \varsigma(this)(Serve(get); Serve(s); thisActivity.s(s_v); s_v)
\end{aligned}$$

Example of usage

$$Cell.s([]); Cell.set([x = 2]); Cell.get()$$

5. Extended Join Services

$$Join((m_{11}, m_{12}, \ldots, m_{1n_1}), (m_{21}, \ldots, m_{2n_2}), \ldots (m_{k1}, \ldots, m_{kn_k}))$$

$$\begin{aligned}
Join((m_1, m_2), (m_1, m_3)) \triangleq\ & let\ served = false\ in \\
& Repeat \\
& \quad if\ (inQueue(m_1) \wedge inQueue(m_2))\ then \\
& \qquad (Serve(m_1);\ Serve(m_2);\ served := true) \\
& \quad else\ if\ (inQueue(m_1) \wedge inQueue(m_3))\ then \\
& \qquad (Serve(m_1);\ Serve(m_3);\ served := true) \\
& Until(served = true)
\end{aligned}$$

Migration

Simulates the migration: makes the current activity forward the requests to a newly created activity.

Syntax

$$thisActivity.Migrate()$$

Encoding

$$\begin{aligned} Migrate \triangleq \varsigma(this)\, let\ newao = Active(this, sevice)\ in \\ (CreateForwarders(newao); FifoService) \end{aligned}$$

$$CreateForwarders(newao) \triangleq \forall m_j,\ m_j \Leftarrow \varsigma(x,y) newao.m_j(y)$$

Groups

Entity containing several objects that can be accessed as a single one.

Passive Groups

Syntax

$$Group(a_k^{k\in 1..l})$$

Reduction Rules

$$\mathcal{R} ::= \ldots \mid Group(\iota_k, \mathcal{R}, b_{k'})^{k\in 1..m-1, k'\in m+1..l}$$

Store group:

$$\frac{\iota \notin dom(\sigma)}{(Group(\iota_k)^{k\in 1..l}, \sigma) \to_G (\iota, \{\iota \mapsto Gr(\iota_k)^{k\in 1..l}\} :: \sigma)}$$

Field access:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.l_i], \sigma) \to_G (Group(\iota_k.l_i)^{k\in 1..l}, \sigma)}$$

Field update:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.l_i := \iota'], \sigma) \to_G (Group(\iota_k.l_i := \iota')^{k\in 1..l}, \sigma)}$$

Invoke method:

$$\frac{\sigma(\iota) = Gr(\iota_k)^{k\in 1..l}}{(\mathcal{R}[\iota.m_j(\iota')], \sigma) \rightarrow_G (Group(\iota_k.m_j(\iota'))^{k\in 1..l}, \sigma)}$$

Active Groups

Syntax

$$ActiveGroup(a_1, \ldots, a_n, m)$$

Encoding

$$ActiveGroup(a_1, \ldots, a_n, m) \triangleq Group(Active(a_1, m), \ldots, Active(a_n, m))$$

Components

Primitive Component

A *primitive component* is defined from an activity α, a set of *server interfaces* (SI, a subset of the served methods), and a set of *client interfaces* (CI, references to other activities contained in fields):

$$SI_i \subseteq \bigcup_{M \in \mathcal{M}_{\alpha P_0}} M$$

$$PC ::= C_n < a, srv, \{SI_i\}^{i \in 1..k}, \{CI_j\}^{j \in 1..l} >$$

Composite Component

A composite component is a set of components (either primitive (PC) or composite (CC)) exporting some server interfaces (some SI_i), some client interfaces (some CI_j), and connecting some client and server interfaces (defining a partial binding (CI_i, SI_j)). Such a component is given a name C_n. CC is a composite component and C either a primitive or a composite one:

$$\begin{aligned} CC ::= C_n \ll \;& C_1, \ldots, C_m; \{(C_{i_p}.CI_{j_p}, C_{i'_p}.SI_{j'_p})\}^{p \in 1..k}; \\ & \{C_{i_q}.CI_{j_q} \to CI_q\}^{q \in 1..l}; \{C_{i_r}.SI_{j_r} \to SI_r\}^{r \in 1..l'} \gg \end{aligned}$$

$$C ::= PC | CC$$

where each C_i is the name of one included component C_i $(i \in 1..m)$, supposed to be pairwise distinct; each exported SI is only bound once to an included component, and each internal client interface $(C_i.CI_j)$ appears at most one time:

$$\forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \begin{cases} p \neq p' \Rightarrow C_{i_p}.CI_{j_p} \neq C_{i_{p'}}.CI_{j_{p'}} \\ q \neq q' \Rightarrow C_{i_q}.CI_{j_q} \neq C_{i_{q'}}.CI_{j_{q'}} \\ C_{i_p}.CI_{j_p} \neq C_{i_q}.CI_{j_q} \\ r \neq r' \Rightarrow SI_r \neq SI_{r'} \end{cases}$$

Deterministic Primitive Component (DPC)

A DPC is a primitive component defined from an activity α, such that server interfaces SI are disjoint subsets of the served method of the active object of α such that every $M \in \mathcal{M}_{\alpha P_0}$ is included in a single SI_i:

$$\begin{cases} \forall i, k,\, i \neq k \Rightarrow SI_i \cap SI_k = \emptyset \\ \forall M \in \mathcal{M}_{\alpha P_0},\, \forall M_1 \subseteq M,\, \forall M_2 \subseteq M \ (M_1 \subseteq SI_i \wedge M_2 \subseteq SI_j) \Rightarrow i = j \end{cases}$$

Deterministic Composite Component (DCC)

A DCC is

- either a DPC,
- or a composite component connecting some DCCs such that the binding between server and client interfaces is one to one. More precisely the following constraints must be added to the ones of Definition 14.2:

$$\begin{cases} \text{Each } C_i \text{ is a DCC} \\ \forall p, p' \in 1..k, \forall q, q' \in 1..l, \forall r, r' \in 1..l' \begin{cases} p \neq p' \Rightarrow C_{i'_p}.SI_{j'_p} \neq C_{i'_{p'}}.SI_{j'_{p'}} \\ r \neq r' \Rightarrow C_{i_r}.SI_{j_r} \neq C_{i_{r'}}.SI_{j_{r'}} \\ C_{i'_p}.SI_{j'_p} \neq C_{i_r}.SI_{j_r} \\ q \neq q' \Rightarrow CI_q \neq CI_{q'} \end{cases} \end{cases}$$

Caromel · Henrio

A Theory of Distributed Objects

Distributed and communicating objects are becoming ubiquitous. In global, Grid and peer-to-peer computing environments, extensive use is made of objects interacting through method calls. So far, no general formalism has been proposed for the foundation of such systems.

Caromel and Henrio are the first to define a calculus for distributed objects interacting using asynchronous method calls with generalized futures, i.e., wait-by-necessity – a must in large-scale systems, providing both high structuring and low coupling, and thus scalability. The authors provide very generic results on expressiveness and determinism, and the potential of their approach is further demonstrated by its capacity to cope with advanced issues such as mobility, groups, and components.

Researchers and graduate students will find here an extensive review of concurrent languages and calculi, with comprehensive figures and summaries.

Developers of distributed systems can adopt the many implementation strategies that are presented and analyzed in detail.

ISBN 3-540-20866-6

springeronline.com