HAMDI YAHYAOUI

ACCELERATION AND SEMANTIC FOUNDATIONS OF EMBEDDED JAVA PLATFORMS

Thèse présentée
à la Faculté des études supérieures de l'Université Laval
dans le cadre du programme de doctorat en informatique
pour l'obtention du grade de Philosophiæ Doctor (Ph.D.)

FACULTÉ DES SCIENCES ET DE GÉNIE
UNIVERSITÉ LAVAL
QUÉBEC

2006

Abstract

With the advent and the rising popularity of wireless systems, there is a proliferation of small internet-enabled devices (e.g. PDAs, cell phones, pagers, etc.). In this context, Java is emerging as a standard execution environment due to its security, portability, mobility and network support features. In particular, J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) is now recognized as the standard Java platform in the domain of mobile wireless devices. An important factor that has amplified the wide industrial adoption of J2ME/CLDC is the broad range of Java based solutions that are available in the market. All these factors made Java and J2ME/CLDC an ideal solution for software development in the arena of embedded systems. A successful deployment of Java on these devices relies on a fast and lightweight execution environment. Our research comes to provide a practical and a theoretical vision about possible solutions to design, implement and validate optimization techniques. More precisely, the research results that led to reach this objective are the following:

1. The design, implementation and evaluation of dynamic acceleration techniques: we have designed and implemented a dynamic selective compiler. This compiler speeds up the execution of embedded Java applications by a rate of 400%. Moreover, we have designed other acceleration techniques for the interpretation and the method call mechanisms.

2. The elaboration of a concurrent denotational semantic model that extends the resource pomsets semantics of Gastin and Mislove with unbounded non-determinism. This model is intended to be accommodated to JVML/CLDC (the bytecode language) and to be used for proving the correctness of the optimizations of JVML/CLDC programs.

3. A case study that shows how this semantic model can be embedded in the proof assistant Isabelle in order to validate optimizations of JVML/CLDC programs.

Résumé

De nos jours, nous assistons à une croissance fulgurante des réseaux sans fil et des systèmes embarqués (cellulaires, assistants digitaux, etc.). Dans ce contexte, Java a connu une popularité grandissante comme étant un environnement d'exécution standard grâce à ses caractéristiques intrinsèques comme la sécurité, portabilité et mobilité. Plus précisément, J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) est devenue une plate-forme standard dans le domaine des systèmes embarqués. En effet, l'important déploiement des téléphones Java a permis une large adoption de cette plate-forme. Le succès de celle-ci nécessite l'existence d'un environnement qui permet une exécution rapide des applications Java. C'est dans ce cadre précis que s'inscrit notre recherche. Notre objectif primordial est de concevoir, implanter et fournir une base formelle pour valider des techniques d'accélération de Java pour les systèmes embarqués. Les principaux résultats ayant contribué à l'atteinte de cet objectif sont les suivants :

1. La conception, l'implantation et l'évaluation d'un compilateur léger et rapide pour l'accélération de l'exécution des applications Java dans les systèmes embarqués. Ce compilateur accélère la machine virtuelle embarquée KVM qui vient avec J2ME/CLDC par un facteur de 4. D'autres techniques d'accélération de l'interprétation et du mécanisme d'appel de méthodes ont été réalisées.

2. L'élaboration d'un modèle sémantique dénotationnel qui étend le modèle resource pomsets de Gastin et Mislove au non-déterminisme non borné. Ce modèle est conçu pour spécifier la sémantique du langage JVML/CLDC (langage bytecode) et aussi pour valider les optimisations de programmes JVML/CLDC.

3. Une étude de cas montrant comment ce modèle peut être embarqué dans l'assistant de preuves Isabelle pour des fins de validation semi-automatique des optimisations de programmes JVML/CLDC.

Acknowledgments

I'm very indebted to Pr. Nadia Tawbi and Pr. Mourad Debbabi for their advices, ideas and efforts to ensure a continuous supervision of this thesis. Their excellent supervision helped me in preserving enthusiasm and motivation to finish this thesis. They deserve all my acknowledgements.

I would like to thank Pr. Mohamed Mejri who has read carefully an earlier version of this thesis and gave thorough comments.

I would like to thank Pr. Kamel Adi and Pr. Marc Frappier who accepted to review this thesis. Their questions, remarks and suggestions will be useful for producing the final version of this thesis.

I would like also to express all my acknowledgements to my colleagues at the Languages, Semantics and Formal Methods Group (LSFM) at Laval University: Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi and Sami Zhioua who shared with me interesting scientific discussions.

Many thanks to Concordia Institute for Information Systems Engineering (CIISE) staff who provided for me an outstanding environment to work. Particularly, I would like to thank Computer Security Laboratory (CSL) members: Luay Alawneh, Chadi Assi, Mourad Azzam, Nadia Belblidia, Ali Bensam, Djamel Benredjem, Hassan Issa, Yosr Jarraya, Marc-André Laverdière, Mona Mehrandish, Hadi Otrok, Sujoy Ray, Mohamed Salah, Payam Shahi, Andrei Soeanu, Syrine Tlili, Anil Venkataiahgari for the grateful moments we shared at CIISE. I'm also grateful to Mrs Lynda Goulet and Rachel Lapierre from the secretariat of the computer science and software engineering department at Laval University. They are so kind and helpful to all the students.

Finally, I would like to thank my friends Nafaâ Jabeur, Nabil Sahli, Mahjoub Langar and Walid Ali for their encouragements, support and solidarity.

To my parents
To my wife and sisters
To my friends and colleagues

Contents

Abstract ii

Résumé iii

Acknowledgments iv

Contents vi

List of Tables xi

List of Figures xii

1 Introduction 1

1.1 Introduction and Motivations 1

1.2 Research Issues 3

1.2.1 Dynamic Compilation for Embedded Java Platforms 3

1.2.2 Optimizations Validation 6

1.3 Objectives 7

1.4 Methodology 8

1.4.1 Reverse Engineering of the KVM 8

1.4.2 State of the Art of Dynamic Compilers 8

1.4.3 A Dynamic Compilation Technique for the Acceleration of Embedded Java Platforms 8

1.4.4 A Semantic Model for True Concurrency with Unbounded Non-Determinism 9

1.4.5 A Case Study for Validating Optimizations Using our Semantic Model 10

1.5 Contributions 11

1.6 Thesis Structure 12

I Embedded Java Platforms Acceleration 13

2 The Java Technology 14

2.1 Introduction 14
2.2 Java 2 Server Edition (J2SE) Platform 15
2.2.1 Java Compilation 15
2.2.2 Java Interpretation Mechanism 17
2.2.3 Java Verification 19
2.3 Java 2 Micro Edition 20
2.3.1 Kilo Virtual Machine (KVM) 21
2.3.2 Mobile Information Device Profile (MIDP) 28
2.4 KVM Reverse Engineering 28
2.4.1 Static Analysis of the KVM 28
2.4.2 Tuning KVM Performance 29
2.4.3 Benchmarking the KVM on Embedded Platforms 30
2.5 Conclusion 30

3 State of the Art of Java Acceleration Techniques 34
3.1 Introduction 34
3.2 Taxonomy of Acceleration Techniques 34
3.2.1 Hardware Acceleration Techniques 35
3.2.2 Ahead of Time Compilation (AOT) 37
3.2.3 Dynamic Compilation 38
3.2.4 General Optimizations 42
3.3 Accelerated Embedded JVMs 46
3.3.1 Kilo JIT (KJIT) 47
3.3.2 Sun CLDC HotSpot (Monty) 48
3.3.3 Insignia Embedded Virtual Machine (EVM) 48
3.3.4 Esmertec Jbed Micro Edition CLDC (Jbed) 48
3.3.5 Acunia Embedded Virtual Machine (Wonka) 49
3.4 Accelerated Conventional JVMs 49
3.4.1 Sun HotSpot 49
3.4.2 IBM JIT Compiler 53
3.4.3 JUDO JIT Compiler 57
3.4.4 LaTTe 60
3.4.5 Aware Just In Time (AJIT) Compilation System 62
3.4.6 Jalapeño Dynamic Optimizing Compiler 64
3.5 Dynamic Compilation Challenges in an Embedded Context 67
3.6 Conclusion 68

4 E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines 70
4.1 Introduction 70
4.2 Architecture 71
4.2.1 Interpreter 72

4.2.2 Native Code Execution Component 73
4.2.3 Profiler 73
4.2.4 One-Pass Compiler 73
4.2.5 Cache Manager 73
4.3 Design 74
4.3.1 Compilation Strategy 74
4.3.2 Switching Mechanism 84
4.3.3 Threads Management 86
4.3.4 Garbage Collection Issues 87
4.4 Implementation and Results 89
4.5 Conclusion 90

5 A Synergy Between Efficient Interpretation and Dynamic Compilation **93**
5.1 Introduction 93
5.2 Compiling by Code Reuse 94
5.2.1 Generated Native Threaded Interpreter 96
5.2.2 Reusing Codelets for Dynamic Compilation 100
5.2.3 Smooth Switching Mechanism 102
5.2.4 Scenario 104
5.3 Enhancing Interpretation by Method Call Acceleration 105
5.3.1 Method Lookup Mechanism 106
5.3.2 Method Lookup Acceleration 108
5.3.3 Caching 111
5.3.4 Implementation & Results 112
5.4 Conclusion 113

II Semantic Foundations of JVML/CLDC **115**

6 Concurrency Models **116**
6.1 Introduction 116
6.2 Mathematical Definitions 118
6.2.1 Some Notions about Domain Theory 118
6.2.2 Some Notions about Category Theory 119
6.2.3 Some Notions about Transfinite Numbers 120
6.3 Operational Models 120
6.4 Denotational Models 121
6.4.1 Failure Sets 121
6.4.2 Acceptance Trees 122
6.4.3 Event Structures 123

6.4.4 Resource Trace Model 123
6.4.5 Resource Pomsets Model 133
6.5 Our Strategy 139
6.5.1 Non-Determinism 140

7 A Semantic Model for True-Concurrency with Unbounded Non-Determinism 143
7.1 Introduction 143
7.2 Language Syntax 146
7.3 Process Space 148
7.3.1 Dependence Maps 148
7.3.2 Labelled Dependence Maps 150
7.3.3 Deterministic Processes 151
7.3.4 Space of Non-Deterministic Processes 155
7.4 Algebraic Properties 155
7.5 Semantics 156
7.5.1 Semantic Interpretation Function 156
7.5.2 Non-Deterministic Choice 157
7.5.3 Strict Sequential Composition 158
7.5.4 Hiding 161
7.5.5 Weak Sequential Composition 163
7.5.6 Parallel Composition 169
7.5.7 Recursion 177
7.6 Details about the Construction of the Space $\mathbb{M}$ 180
7.7 Algebraic Properties 182
7.7.1 Definition of a Co-induction Principle over the Space $\mathbb{M}$ 182
7.7.2 Algebraic Properties: Claims and Proofs 183
7.8 Monotonicity Proofs 189
7.8.1 Monotonicity of the Non-Deterministic Choice 191
7.8.2 Monotonicity of the Strict Sequential Composition 191
7.8.3 Monotonicity of Hiding 195
7.8.4 Monotonicity of the Weak Sequential Composition 198
7.8.5 Monotonicity of the Parallel Composition 205
7.8.6 Fixed Point Existence 212
7.9 Healthiness Conditions 216
7.9.1 First Healthiness Condition Verification 216
7.9.2 Second Healthiness Condition Verification 220
7.10 Conclusion 222

8 Towards A Framework for Validating Optimizations of JVML/CLDC Programs 223
8.1 Introduction 223

8.2 Related Work 223
8.3 Isabelle at a Glance 225
8.3.1 Theories 225
8.3.2 Datatypes 225
8.3.3 Function Declaration 225
8.3.4 Tactics 226
8.4 Embedding the Semantic Model 226
8.5 JVML/CLDC Subset Syntax 228
8.6 A Case Study about the Validation of Optimizations of JVML/CLDC Programs 230
8.6.1 Constant Propagation 230
8.6.2 Dead Assignment Elimination 232
8.6.3 Common Subexpression Elimination 232
8.6.4 Denotational Semantics 234
8.7 Conclusion 245

9 Conclusion **246**

Bibliography **249**

List of Tables

2.1 Exception table 18

3.1 Dynamic call optimization in the presence of dynamic class loading . . 54
3.2 A Jalapeño optimized HIR 65

4.1 Executable file footprint overhead 89

5.1 Context-free bytecodes 97
5.2 Context-dependent bytecodes 98

List of Figures

1.1 Morris approach for compiler correctness 6
1.2 Wand paradigm for proving compiler correctness 7

2.1 A Java program 16
2.2 Java bytecodes 16
2.3 try/catch code 18
2.4 Compiled program 19
2.5 Possible configurations 20
2.6 Profile Architecture 21
2.7 Interaction between some components of the KVM 22
2.8 Romizing process 23
2.9 Interpreter 25
2.10 KVM stack layout 26
2.11 Thread synchronization in KVM 27
2.12 Snapshot of KVM analysis in Imagix 31
2.13 KVM scores on an IPAQ 32
2.14 Partial view of the KVM call graph 32
2.15 KVM tuning 33

3.1 Class hierarchy example 50
3.2 HotSpot client performance 53
3.3 IBM-JIT compiler architecture 55
3.4 Code translation in IBM-JIT 56
3.5 IBM-JIT performance 57
3.6 JUDO components 58
3.7 JUDO optimizing compiler structure 58
3.8 JUDO performance 60
3.9 LaTTe architecture 61
3.10 LaTTe performance 62
3.11 VRA annotation for `iaload` opcode 63
3.12 AJIT performance 63
3.13 Jalapeño architecture 64

4.1 E-Bunny architecture 72
4.2 Context saving 75
4.3 Context restoration 76
4.4 iload bytecode translation 76
4.5 iload on the native stack 77
4.6 ifeq bytecode translation 78
4.7 putfield bytecode translation 80
4.8 invokevirtual bytecode translation 82
4.9 athrow bytecode translation 83
4.10 Interpreted to native mode switching algorithm 85
4.11 Switching to the native mode 85
4.12 Native to interpreted mode switching algorithm 86
4.13 Multi-threading in E-Bunny 87
4.14 Garbage collection algorithm modifications 88
4.15 CaffeineMark scores of KVM and E-Bunny with GCC 89
4.16 CaffeineMark midlet ran by KVM and by E-Bunny 90
4.17 Snapshot of an E-Bunny generated code 92

5.1 Generated native threaded interpreter structure 99
5.2 Codelet for `aload_3` 99
5.3 Interpreter codelet generation 100
5.4 Native code generation scheme 101
5.5 Compilation of a bytecode 101
5.6 The unified stack layout 102
5.7 A lightweight interpreter/compiler mode switching mechanism 103
5.8 Prologue 104
5.9 Epilogue 105
5.10 Bytecode sequence under compilation 105
5.11 A scenario of the proposed compilation by code reuse technique 106
5.12 Method hash table construction algorithm 109
5.13 Original lookup algorithm 110
5.14 Optimized lookup algorithm 110
5.15 Threaded interpreter performance 112
5.16 Typical example for our Optimization technique 113
5.17 Execution time acceleration for a typical example 114
5.18 Trade-Off between footprint and collisions 114

6.1 Morris Approach to Compiler Correctness 117
6.2 Wand Paradigm for Proving Compiler Correctness 117
6.3 Example of a resource trace 126
6.4 Example of a concatenation of two resource traces 127

6.5 A restriction of a process execution to a set of resources 130
6.6 The parallel composition of two processes 132
6.7 Example about strict sequential composition 136
6.8 Example about hiding a process 137

7.1 Example of a graphical representation of a dependence map 150
7.2 An example illustrating the operator \ on dependence maps 153
7.3 A strict sequential composition of two finite processes 160
7.4 A strict sequential composition of a finite and an infinite Process 161
7.5 A hiding of events using some resource set from a process 163
7.6 A hiding of events using a resource set from a recursive process that results into a divergence 163
7.7 Weak sequential composition of two processes 166
7.8 Another example of weak sequential composition of two processes . . . 167
7.9 Example about the emergence of transfinite dependencies in the presence of weak sequential composition of two processes 169
7.10 A parallel composition of two finite processes 174
7.11 A parallel composition of two infinite processes 176
7.12 A parallel composition of two non-terminated processes 178
7.13 The semantics of the process P 179
7.14 The semantics of the process Q 180
7.15 The semantics of the process T 180
7.16 Cone Existence Diagram 181

8.1 Development Environment 227

Chapter 1

Introduction

1.1 Introduction and Motivations

With the advent and the rising popularity of wireless systems, there is a proliferation of small internet-enabled devices (e.g. PDAs, cell phones, pagers, etc.). In this context, Java [55, 72] is emerging as a standard execution environment due to its security, portability, mobility and network support features. In particular, J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Device Configuration) [79] is now recognized as the standard Java platform in the domain of mobile wireless devices. It gained big momentum and is now standardized by the Java Community Process (JCP) and adopted by many standardization bodies such as 3GPP and MEXE. Another factor that has amplified the wide industrial adoption of J2ME/CLDC is the broad range of Java based solutions that are available in the market. In fact, the number of Java-enabled phones that are deployed in the market is estimated to be more than 1 billion [84]. All these factors made Java and J2ME/CLDC an ideal solution for software development in the arena of embedded systems. J2ME/CLDC consists of three layers: a lightweight virtual machine (e.g. KVM or Kilobyte Virtual Machine), a configuration (Connected Limited Device Configuration) and a profile (e.g. MIDP or Mobile Information Device Profile). Wireless devices have two main limitations:

- Memory limitations: most of wireless devices are very resource-constrained. Typical devices such as phones have less than 512 KB of RAM.
- Power limitations: the battery life of these devices is short. The processors are very slow (e.g. 206 MHz for IPAQ H3600) in order to minimize power consump-

tion.

The "Write once, run everywhere" Java paradigm is considered as a programming revolution. The main features of the Java revolution like portability, reliability and security make the deployment of Java in the wired world a very successful story. However, Java application execution is very slow in embedded devices. In fact, the KVM runs from 30 to 80% of the speed of a conventional Java Virtual Machine (JVM) [141]. This fact leads to a real need for the acceleration of Java for embedded systems.

Java acceleration techniques are classified in two categories:

- Hardware acceleration techniques: Java is executed directly by hardware processors or co-processors. This entails a significant speedup of execution (up to 30 times [91]). However, their use comes with a high price in terms of power consumption. This energy issue is really damaging especially in the case of low end mobile devices.
- Software acceleration techniques: there are two subcategories of software acceleration techniques:
  - Static acceleration techniques: these techniques are applied before execution. By using heavy static analysis and optimizations, the execution of Java applications is considerably enhanced (up to 10 times [134]). However, these techniques entail an important memory footprint [1], which is unacceptable in an embedded context. Moreover, they assume that the application code is always available, which is not true in a dynamic environment like the JVM or KVM. In fact, the code can be loaded dynamically from a distant site.
  - Dynamic acceleration techniques: these techniques are applied dynamically. This allows to profit from the available dynamic information about the code. Dynamic compilation[2] is the most prominent dynamic acceleration technique. In fact, this technique is able to achieve significant speedup (up to 20 times). However, traditional dynamic compilers like Sun HotSpot [78] or IBM Jalapeño [8] consume a lot of power and memory space to perform analysis. This is unacceptable in an embedded context.

Few research initiatives [62, 84, 113] targeted the acceleration of Java for embedded platforms. These initiatives are led by some industrial companies for commercial goals.

[1]Overhead in terms of memory

[2]Dynamic compilation consists in translating dynamically a source code to native code

What is published is restricted to white papers about the general features of the accelerated JVMs. This is the main reason for having a poor related work for this research field.

All these facts have motivated us to elaborate a research thesis about Java acceleration in embedded systems. The main challenge of this research is to design fast, lightweight and correct optimization techniques that can be applied and integrated in embedded Java platforms such as J2ME. More accurately, we investigated dynamic acceleration techniques and particularly dynamic compilation for embedded Java platforms.

1.2 Research Issues

In the sequel, we present the research issues that we faced at the practical and theoretical levels.

1.2.1 Dynamic Compilation for Embedded Java Platforms

When we started this research, there was a lack of academic initiatives about the acceleration of embedded Java platforms. Actually, only industrial companies led industrial projects to accelerate embedded Java platforms (mainly J2ME/CLDC). For instance, Sun [77], Insignia [62], Esmertec [46] and few companies worked and are still working on this topic without publishing details about their research. They provide just white papers or advertising documents about these projects. The ultimate goal of their projects is to produce enhanced versions of the KVM or devise new accelerated embedded virtual machines.

This entailed the need for studying the dynamic compilation of Java in desktop and server systems. This study was a useful step to know in deep the proposed solutions and see if they can be applied in an embedded context.

We faced many practical issues when we were in the process of designing a dynamic compilation technique for embedded Java platforms and particularly for the KVM. We present hereafter these issues.

Code Translation

In the literature, many dynamic compilers build an intermediate representation for the Java code. Flow-based analyses are performed on this representation. The aim of these analyses is to optimize the code (e.g. constant propagation, dead code detection) and to compute some information about variables (e.g. liveness information) that is useful to generate a native code of good quality.

The use of heavy flow-based analysis techniques is not convenient for memory-constrained embedded platforms (having an available memory that is less than 512 KB). Moreover, these techniques perform many passes so they are not efficient at the compilation side. Hence, a tradeoff between compilation time and code quality should be established in the embedded context.

Compilation Unit

The first designed dynamic compilation strategy consists in compiling all the loaded code. This strategy entails an important footprint, which is unacceptable. Accordingly, dynamic selective techniques have emerged. The selection is based on the detection of the frequently executed code known as *hotspot* code. In fact, a compilation unit can be a method or a set of instructions. Each of these compilation units has its advantages and disadvantages. The design of a lightweight hotspot detection technique and the choice of the compilation unit are two important practical issues.

Communication between the KVM Interpretation and Native Execution

Generating native code from a JVML/CLDC[3] program is a critical step that prepares for its execution. In a mixed mode execution (interpreted/compiled), a communication between the Java world (the KVM interpreter) and the native world can occur frequently. Traditional Java Virtual Machines (JVMs) use a Java Native Interface (JNI) to invoke native code from a Java code. This mechanism is not implemented in embedded JVMs, like the KVM, because it is very heavy. A call from an interpreted method into native code requires setting the call context (e.g. parameter transfer from the Java stack to the native stack, local variables setting, etc.). A design of an efficient

[3]JVML/CLDC is the binary language for embedded Java

and lightweight Java to native interface, that enables the switch between compiled and interpreted code (and vice versa), is one of compilation issues.

Exception Mechanism Translation

The KVM exception propagation mechanism consists in looking for a handler of a thrown exception in the method that raises it. When no handler is found, the search continues in the calling method. Dynamic compilation should preserve the exception propagation semantics not only between interpreted methods but also between interpreted and compiled methods in a lightweight manner.

Threading

The KVM uses a round-robin algorithm to perform thread rescheduling after the execution of some instructions. All active threads are kept in a circular linked list. A time slice is assigned to each thread. A zero-value of this counter triggers thread rescheduling. Hence, the next thread in the circular list is selected for execution. Since the KVM interpreter cannot manipulate the native code, a design of a thread rescheduling mechanism between the native world and the Java world (and vice versa) is required. The issue here is to design compact data structures that are required to preform thread rescheduling in a mixed mode execution.

Cache Management

A dynamic compiler requires memory buffers to hold translated native code that it produces. It is known that Java instructions (Java bytecodes[4]) are much smaller than native instructions. Actually, the code generated by a dynamic compiler is generally 4 to 5 times [84] the original code size. For embedded platforms, this code should be saved in a dedicated cache. Since the cache size is limited, the elements of the cache have to be replaced when the cache is full and a new compiled code has to be added to the cache. Consequently, a cache replacement policy is required.

[4]The binary instructions of Java

$$\begin{tikzcd}
P_1 \arrow[r, "compile"] \arrow[d, "semantics_{source}"'] & P_2 \arrow[d, "semantics_{target}"] \\
Semantics_1 \arrow[r, "encode"] & Semantics_2
\end{tikzcd}$$

Figure 1.1: Morris approach for compiler correctness

1.2.2 Optimizations Validation

Establishing the semantic correctness of an optimization technique consists in proving that the optimization preserves the semantics i.e. the original program and the optimized one are semantically equivalent. This entails the elaboration of one semantics if the original and optimized programs are defined in the same language. In the case of dynamic compilation, we are in the presence of two languages[5]: the source language and the target language. This means that two semantics are needed.

In the literature on programming languages, many researchers used the method introduced by Morris in [90], further promoted in [130], to establish the correctness of a compilation/optimization process. This approach advocates the use of algebraic data types and algebraic semantics to capture the optimization correctness as the following equation:

$$encode(semantics_{source}(P_1)) = semantics_{target}(compile(P_1))$$

This amounts to the commutation of the diagram reported in Figure 1.1. Later, this approach has been accommodated to use an operational semantics style as what Stephenson proposed in [122] or a denotational semantics style as what Wand proposed in [133]. In a denotational semantics setting, the correctness of the compiler is expressed as the equality of the denotation of the source program and the denotation of its translation. This paradigm for proving compiler correctness is outlined in Figure 6.2.

To establish the correctness of our dynamic compilation or any other optimization that can be performed on JVML/CLDC programs, we have to provide first a semantic model for JVML/CLDC. Particularly, since JVML/CLDC is a concurrent language, we have to select or elaborate a concurrency semantic model for this language. This is

[5]The source code is JVML/CLDC and the target code is the binary language of Intel processors.

$$\begin{tikzcd}
P_1 \arrow[r, "compile"] \arrow[dr, "semantics_{source}"'] & P_2 \arrow[d, "semantics_{target}"] \\
& \text{Intermediate Language}
\end{tikzcd}$$

Figure 1.2: Wand paradigm for proving compiler correctness

another big challenge for this research. Moreover, the choice between the operational or denotational strategy is one of the important decisions in this research project. The reasons underlying our decision are provided later.

1.3 Objectives

The main intent of this thesis is to contribute at two levels: acceleration and semantic foundations of embedded Java platforms. We target the design of semantically-correct acceleration techniques for embedded Java platforms. Particularly, we are interested in dynamic acceleration techniques since static compilation is not adequate in an environment where applications can be loaded dynamically. More accurately, our objectives are threefold:

- Design, implement and evaluate dynamic acceleration techniques for embedded Java platforms and particularly for the KVM, which is the defacto standard embedded VM. This practical task allows to enrich the related work on the acceleration of embedded Java platforms.
- Provide a semantic model for JVML/CLDC. The main traits of this model are: provability, compositionnality and readability. By provability, we mean that the model can be a basis for formal validation of optimizations. Compositionality means that the semantics of an expression is defined in terms of the semantics of its sub-expressions. Besides, the semantics should be abstract and clear.
- Show that our semantic model can be embedded in a theorem proving tool and used in order to validate several optimizations. The use of a proof assistant allows to provide correct and machine-checked proofs.

1.4 Methodology

In the sequel, we detail our methodology.

1.4.1 Reverse Engineering of the KVM

We spent an important time to understand the inner working of the KVM components and to detect where it is possible to enhance execution. We used a static analysis tool (Imagix) to understand the interaction between KVM components. Moreover, we used a dynamic profiling tool (Intel VTune) to detect critical functions in the KVM. Our first conclusion is that the interpretation mechanism represents a high percentage of the overall execution with respect to the other mechanisms (loading, verification, etc.). This allowed us to know that it is more relevant to concentrate our efforts on the acceleration of the interpretation mechanism than the other ones. Thus, we decided to enhance the interpretation mechanism by a dynamic strategy. The reasons for excluding the static strategy are mentioned before.

1.4.2 State of the Art of Dynamic Compilers

The lack of academic research initiatives on the acceleration of embedded Java platforms is the main reason for the study of several dynamic compilers for desktop and server systems. This study enabled us to know the proposed techniques to accelerate Java in these systems. We have elaborated a detailed description of the architectures of well-known dynamic compilers. In addition, we have studied their performance, their strengths and weaknesses. Thanks to this study, we elaborated a strategy for achieving a successful dynamic compilation technique for embedded Java platforms.

1.4.3 A Dynamic Compilation Technique for the Acceleration of Embedded Java Platforms

We give hereafter a brief overview of our dynamic compilation strategy:

- The dynamic compilation is selective (restricted to hotspots, which are frequently called methods).

- To keep the memory footprint low, we do not use any heavy static analyses or representations. The code translation is performed in one-pass. This avoids spending much time in generating native code and offers a reasonable speedup.
- There is no register allocation. More precisely, we avoid the use of heavy register allocation algorithms (such as graph coloring or algorithms that use liveness information). The time spent in register allocation will be saved. Only optimizations that can be performed in one-pass are allowed.
- The generated code should be cached. The cache is small with careful management: a limited space for the cache allows to respect the memory limitations of embedded systems.

The full detailed compilation strategy is explained later in this thesis.

In addition, we come with a dynamic acceleration technique that establishes a synergy between dynamic compilation and interpretation. This technique enhances our compiler by a unified interpretation model in which the compiler and the interpreter are collaborating in a smooth way.

1.4.4 A Semantic Model for True Concurrency with Unbounded Non-Determinism

To comply with the requirements of provability, compositionality and readability, we have decided to elaborate a denotational semantic model that can be accommodated to JVML/CLDC and follow Wand [133] approach to establish optimization correctness. JVML/CLDC is a concurrent language. This requires the selection/elaboration of an adequate concurrent and denotational model.

Concurrency models are classified w.r.t three major criteria [137]: the focus on the state or the behavior, the treatment of parallelism through interleaving or true concurrency and the way by which non-determinism is handled.

Famous denotational models such as failure sets and acceptance trees [19, 58] put focus on the behavior. They are branching models and they give an interleaving meaning to parallelism. In fact, in these models, the parallel composition of two processes is reduced to the possible interleaving between them. This leads to a state explosion problem.

True concurrency models minimize this state explosion by using partial orders. In these models, independent events/actions can be executed simultaneously. Famous true concurrency models such as labelled event structures [135] and labelled transition systems with independence [13] can capture true concurrency but they lack explicit description of it.

Lately, Gastin and Mislove [50] provide an explicit description of true concurrency in a model they called resource pomsets. The model is extensional, which means that it puts more emphasis on the behavior than the system itself. Moreover, it is linear since it does not consider branching points. The main interesting feature of this model is the resources, which play a fundamental role to specify when two events can be executed in parallel. For our research, the resource concept is useful since we aim to provide a semantics that describes in an explicit way the Java synchronization mechanism. For instance, they can be used to denote objects that can be locked by processes when they execute synchronized methods/blocks. Moreover, the use of resources establishes a good connection between actions/events and data.

Unfortunately, the resource pomsets model reduces parallel composition of events sharing the same resource to a deadlock. This is not true in many concurrent languages such as JVML/CLDC. In fact, when two processes are claiming, at the same time, the same synchronized method, the execution interleaves between these processes. This leads to the need for the extension of the resource pomsets model to include non-determinism.

We have extended the resource pomsets model with unbounded non-determinism. The rationale behind choosing an unbounded non-determinism is that this form of non-determinism allows to abstract from implementation details. More reasons about adopting this form of non-determinism are provided later in this thesis.

1.4.5 A Case Study for Validating Optimizations Using our Semantic Model

We provide a case study about validating some optimizations of JVML/CLDC programs using our semantic model. The validation is performed using the theorem prover Isabelle [95]. More precisely, we present an embedding of a subset of JVML/CLDC with its denotational semantics in Isabelle. We also provide a particular discussion about the equivalence relation that can be adopted in proving the semantic equivalence between JVML/CLDC programs. Furthermore, we discuss the semantic equivalence between several JVML/CLDC programs and their optimized versions. The studied optimizations are: constant propagation, dead assignment elimination and common subexpression elimination.

1.5 Contributions

The main contributions of this thesis are the following:

- The design, implementation and evaluation of new dynamic acceleration techniques for embedded Java virtual machines: we have designed and implemented a dynamic selective compiler (called E-Bunny), which speeds up the KVM execution by a rate of 400%. We have also designed an acceleration technique for the method call mechanism. The proposed technique is dynamic, flexible and efficient. Note that this practical work was leaded with some colleagues of the LSFM (Languages, Semantics and Formal Methods) research group who are: Abdelouahed Gherbi, Lamia Ketari, Chamseddine Talhi and Sami Zhioua.
- The design of a concurrent denotational semantic model that extends the resource pomsets semantics of Gastin and Mislove [50] with unbounded non-determinism. More precisely, we provide the semantic interpretation of some useful concurrency operators and a fixpoint semantics of recursion. This model is intended to be accommodated for JVML/CLDC.
- A case study that shows how this semantic model can be embedded in the proof assistant Isabelle in order to validate some optimizations of JVML/CLDC programs.

As a downstream result, we succeeded to publish the following research papers:

- A paper, entitled "E-Bunny: A Dynamic Compiler for Embedded Java Platforms", is published in the Journal of Object Technology (JOT), volume 4, number 1, 2005. The main contribution of this paper is the design and implementation of a fast and lightweight dynamic compiler that accelerates by a rate of 400% the original KVM.

- A paper entitled "A Selective Dynamic Compiler for Embedded Java Virtual Machines Targeting ARM Processors", is published in the International Journal of Science of Computer Programming, volume 59, issues 1-2, 2006. The main contribution of this paper is the porting of E-Bunny to the ARM architecture.
- A paper, entitled "A Synergy between Lightweight Dynamic Compilation and Fast Interpretation", is published in Proceedings of Principles and Practice of Programming in Java (ACM PPPJ'04). The main contribution of this paper is the design and implementation of a fast and lightweight interpreter generation.
- A paper, entitled "Method Call Acceleration in Embedded Virtual Machines", is published in the Workshop of Java in Computation Science (WJCS'2003). The main contribution of this paper is the design and implementation of dynamic, flexible and efficient techniques for accelerating the method call mechanism for embedded Java Virtual Machines.
- A patent submitted to European, Asian and American patent offices. This patent deals with the acceleration of Java Method Call in Virtual Machines.
- A paper about the design of a new semantic model for true concurrency with unbounded non-determinism was under consideration for publication in a conference.

1.6 Thesis Structure

The rest of this document is organized as follows. In the second chapter, we give an overview of the main features of the Java language, J2ME and particularly the KVM. In the third, we present the related work about Java acceleration techniques and give insight into the challenges in an embedded context. The fourth chapter is devoted to the design and implementation of a fast and lightweight dynamic compiler for embedded Java platforms. In the fifth chapter, we give insight into a lightweight compilation technique that extends what we propose in the fourth chapter. We also present a method call acceleration technique that we designed and implemented in the KVM. In the sixth chapter, we present the known concurrency models and put focus particularly on the resource pomsets semantic model. In the seventh chapter, we provide a denotational semantic model for true concurrency with unbounded non-determinism. This model is intended to be accommodated for JVML/CLDC. In chapter eight, we show how our semantic model can be used in the theorem prover Isabelle in order to validate optimizations. Finally, we give some conclusions and possible continuations for this research.

Part I

Embedded Java Platforms Acceleration

Chapter 2

The Java Technology

2.1 Introduction

Java is recognized as a prominent language for the development of mobile code. In fact, Java is a portable, secure and reliable language. The success of Java is due to the following features:

- Strong Typing: each Java construction has a type that is known statically. This allows Java compilers to generate type-safe programs. Moreover, a dynamic verification process allows to check the typing of dynamically loaded Java applications. This verification ensures that properties like control flow, stack and memory safety are satisfied by the verified code.
- Automatic Memory Management: the memory management is transparent to users. In fact, there is no way to manipulate directly the memory. The garbage collector is the component responsible for memory management.
- Platform Independence: Java bytecodes are platform-independent. This allows a high portability. A machine-dependent interpreter, residing in the virtual machine, is responsible for executing Java bytecodes.

In the sequel, we present two Java platforms: Java 2 Server Edition (J2SE) for server and desktop systems and Java 2 Micro Edition (J2ME) for embedded devices.

2.2 Java 2 Server Edition (J2SE) Platform

J2SE provides an environment for the development of Java applications intended for desktops and servers. J2SE comes with Java Application Programming Interface (API) classes and a Java Virtual Machine (JVM). A typical JVM is composed of: a loader, a verifier, an interpreter and a garbage collector. The interpreter executes Java classfiles, which are the result of compiling source Java files by a Java compiler. In the sequel, we give an overview about Java compilation and interpretation mechanisms.

2.2.1 Java Compilation

The target language of the Java compilation is called Java Virtual Machine Language (JVML). It is a machine-independent stack-based language. This format ensures the portability of Java. The compilation of a source Java class provides a classfile that contains Java bytecodes. In the sequel, we present the structure of a classfile and an example illustrating the compilation of a Java program to a JVML one.

Java Classfile Structure

A Java file is compiled into a classfile. Each classfile has a constant pool, a list of fields and a list of methods. A class constant pool contains entries refereing to fields and methods of the class.

A Java method structure is composed of a list of modifiers (e.g static, public), its local variables table, its code (bytecodes) and its exception table.

Each bytecode is composed of an opcode and one or many operands. For instance, the bytecode invokevirtual #5 has invokevirtual as opcode and #5 as operand. This bytecode means that a method that has the same signature as the 5^{th} method in the constant pool of the call receiver (the object on which the call is performed) class is invoked. The call receiver class is resolved statically but the real method to be invoked can figure in a subclass of this class.

Figure 2.1 outlines a simple Java program and Figure 2.2 presents its compiled version.

```
public class HelloWorld
{
        private String Hello;
        public void printHello()
        {
                Hello = "Hello World";
                System.out.println(Hello);
        }

        public static void main(String[] args)

        {
        HelloWorld hw = new HelloWorld();
        hw.printHello();
        }
}
```

Figure 2.1: A Java program

```
Method void printHello()
  0 aload_0
  1 ldc #2 <String "Hello World">
  3 putfield #3 <Field java.lang.String Hello>
  6 getstatic #4 <Field java.io.PrintStream out>
  9 aload_0
  10 getfield #3 <Field java.lang.String Hello>
  13 invokevirtual #5 <Method void println(java.lang.String)>
  16 return
```

Figure 2.2: Java bytecodes

Examples of Opcodes

Hereafter, we present some examples of opcodes:

- invokevirtual: opcode for a virtual (non-static) method invocation.
- getfield: opcode for a non-static field access.
- checkcast: opcode for checking a subtyping relation.
- monitorenter: opcode for locking an object.

A complete description of Java bytecodes can be found in the JVM specification book [72].

2.2.2 Java Interpretation Mechanism

The execution of a Java application starts by loading the system classes and then the classfile given by the user in the command line (and eventually its superclasses). Then, a verification step is performed. The objective of this step is to verify the structure and the well-typing of the loaded classes.

Once the verification is performed, the interpretation step takes place. The interpreter has a main loop that iterates on each bytecode of a called method. First, the main static method of the application is executed. Then, for each called method, a frame is created on the Java stack. This frame contains useful information to restore the calling context. Among these information, we mention:

- A reference to the local variables table (*lp*) of the method,
- A reference to the runtime constant pool (*cp*) of the called method class,
- A reference to the stack (*sp*) that will contain the temporary values created while executing this method,
- A reference to the previous instruction pointer (*PreviousIp*): the instruction pointer of the instruction following the call,
- A reference to the previous stack pointer (*PreviousSp*): the stack operand of the calling method and
- A reference to the previous frame pointer (*PreviousFp*): the frame of the calling method.

A frame destruction is performed when a call returns from a called method to the calling one. The execution continues at the previous instruction pointer. The previous stack pointer will be the current stack and the previous frame pointer will be the current frame.

Java Exceptions

The Java exception mechanism is very convenient to help developers detecting the errors that can occur. An exception is thrown when a violation of some Java safety rules

```
void catchTwo()
}
try {
tryItOut();
}
catch(TestExc1 e) {handleExc(e); }
catch(TestExc2 e) {handleExc(e); }
}
```

Figure 2.3: try/catch code

From	To	Target	Type
0	4	5	Class TestExc_1
0	4	12	Class TestExc_2

Table 2.1: Exception table

happens. For instance, when there is an access to a null pointer or an access to an index that is outside the boundary of an array etc. This exception can be caught and handled in the application. An exception construct is a try/catch/finally, try/catch, or try/finally instruction. Clauses starting with catch represent possible execution continuations after the occurrence of an exception. An exception can be thrown by a throw instruction or by the VM itself. The exception type is used to look for a compatible catch (the parameter of the catch instruction is a super type of the thrown exception type). The execution continuation will be the first catch that satisfies this condition. Any enclosing finally clause must be executed even if there is no catch that handles the thrown exception. Figure 2.3 shows a Java code for a try/catch instruction.

Figure 2.4 shows the bytecodes resulting from the compilation of the Java code presented in Figure 2.3. Since the Java source code contains exception constructs, the Java compiler builds an exception table that describes the possible execution continuations after a possible exception. The exception table is presented in Figure 2.1. It says the following: when an exception is raised between the 0^{th} and the 4^{th} instruction, it can be caught at the 5^{th} instruction if its type is a subtype of TestExc_1. The continuation can be at the 12^{th} instruction when the exception is a subtype of TestExc_2.

```
void catchTwo()
0 aload_0
1 invokevirtual#5
4 return
5 astore_1
6 aload_0
7 aload_1
8 invokevirtual#7
11 return
12 astore_1
13 aload_0
14 aload_1
15 invokevirtual#7
18 return
```

Figure 2.4: Compiled program

Java Threads

Thread management in Java is performed at a software level. Two main thread management mechanisms are provided by a JVM:

- Switching: switching between threads requires saving the context of the current method in the thread structure and loading another method context from the thread to which the execution switches.
- Synchronization: the thread executing a synchronized method locks the call receiver object. Any other thread trying to execute this method, on the same receiver object, is blocked (saved in a wait queue) until the thread owning the lock releases it. When a liberation of the lock occurs, a notification is performed by the virtual machine to awaken blocked threads. The synchronization can be performed also on block of instructions.

2.2.3 Java Verification

Java verification is performed dynamically. It consists of performing data flow analyses on the control flow graph of each loaded method. The intent of these analyses is to ensure that some properties are satisfied by the loaded code. For instance, these anaylses can ensure that no stack overflow will occur during execution and that methods are invoked with the appropriate arguments.

2.3 Java 2 Micro Edition

The Java 2 Micro Edition technology (J2ME) is a Java platform that is dedicated to resource-constrained devices. Two concepts are introduced under the J2ME technology:

- Configuration: due to the wide range of wireless devices, Sun has introduced the configuration concept, which is the combination of a set of APIs, class libraries and a virtual machine, dedicated to a set of similar devices. Two standard configurations are available: the Connected Limited Device Configuration (CLDC) that is intended for smaller wireless devices with less than 512 KB of available memory (cellular phones, pagers, PDAs, etc.) and the Connected Device Configuration (CDC) that is intended for larger wireless devices with at least a few megabytes (2 to 16 Megs) of available memory (Internet TV, gaming consoles, etc.). The CDC configuration includes the C Virtual Machine (CVM), a full J2SE-compliant virtual machine. The CLDC configuration includes the Kilo Virtual Machine (KVM). Figure 2.5 outlines the possible configurations.
- Profile: a profile offers to developers user interface APIs dedicated to a specific set of devices. Mobile Information Device Profile (MIDP) is an example of a profile using the CLDC configuration. The profile architecture is depicted in Figure 2.6.

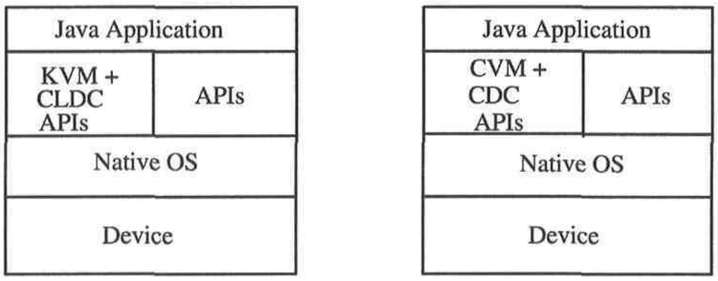


Figure 2.5: Possible configurations

We are particularly interested in studying J2ME/CLDC, which is designed for severely memory constrained-devices (less than 512 KB of available memory). In the sequel, we present the KVM, which is the cornerstone component of J2ME/CLDC.

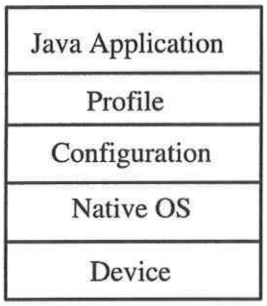


Figure 2.6: Profile Architecture

2.3.1 Kilo Virtual Machine (KVM)

KVM (Kilo Virtual Machine) [81] is Sun's compact Java Virtual Machine technology, intended for small wireless devices (cellular phones, PDAs, etc.). It is a particular implementation of a Java virtual machine meeting the CLDC specification, which is aimed at defining a standard Java platform for small, resource constrained, connected devices.

The KVM is implemented in a portable C language. The core of this VM is about 24,000 lines of code including comments. The KVM executable size varies from 40 to 80 KB depending on the target platform and the used compilation options. KVM performance on embedded devices is not satisfactory. In fact, it is established that KVM runs from 30 to 80% of the speed of a JVM without a dynamic compiler [141].

Due to resource limitations of embedded devices, many features of Java were omitted/redisigned in the KVM. For instance, the floating operations consume a lot of power so they were removed. Notice that recent versions of the CLDC specification (since version 1.1) include the floats. Naturally, this comes with a memory footprint and power cost.

Bytecode verification is one of the JVM features that were redesigned in KVM. In fact, the traditional bytecode verification requires an important memory space that is not convenient for embedded devices. This is due to heavy data flow analyses that are performed dynamically. Java Verification is dispatched in two steps: a static preverification step and a dynamic lightweight verification step. Figure 2.7 outlines the relation between the static (offline) and dynamic (online) verification steps and also the interaction between some of the KVM components.

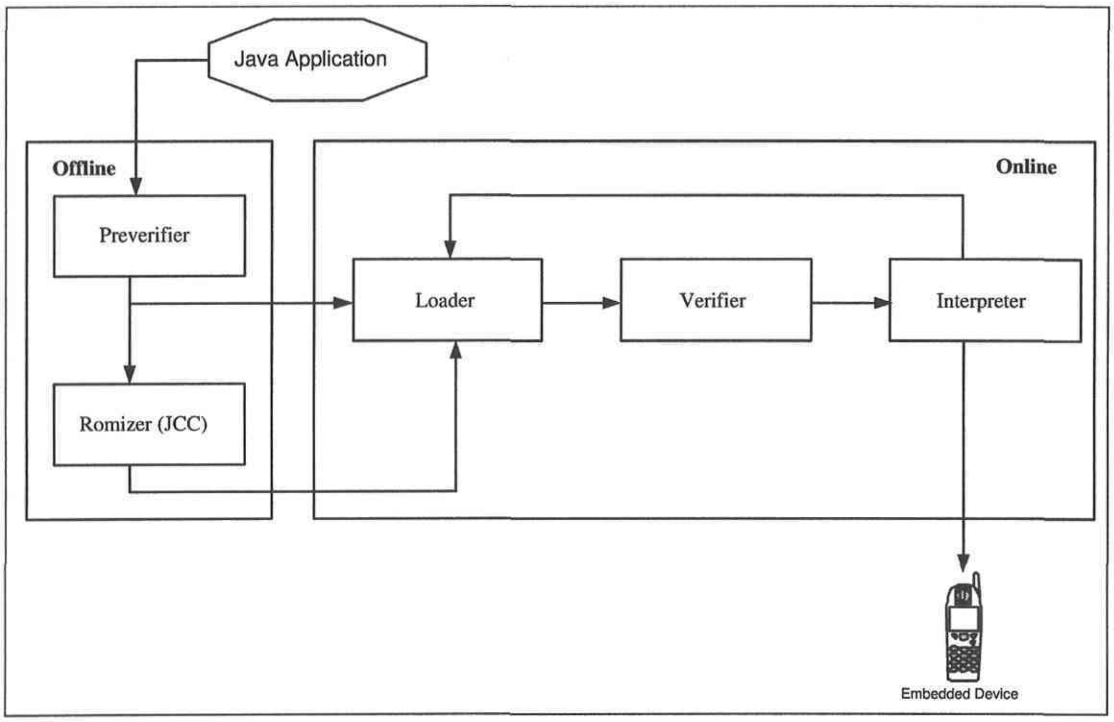


Figure 2.7: Interaction between some components of the KVM

In the sequel, we give an overview about the main KVM components.

Class Loading

The class loading process consists of loading Java system classes and user classes. The result of the loading process is the creation of constant pools of the loaded classes. These constant pools contain information that are used at the execution step such as typing information, fields and methods information.

At run-time, class loading consumes a lot of memory. To avoid the loading overhead, Sun offers in the KVM, a way to pre-load the classes. In fact, a romizing tool translates Java classes into C structures that contain all the information about the compacted classes. These structures are stored in ROM. So, the loading process is accelerated and a considerable RAM memory space is saved. The romizing tool is composed of two components: a Java Filter and a Java Code Compactor (JCC). The Java filter gets as input a Java application and Java APIs then computes the required fields and methods to be used. This allows to avoid loading unnecessary fields and methods. Then, JCC generates a file *romJava.c* that contains the C structures corresponding to the filtered classes. Finally, the object code of this file is linked to the KVM object code. The romizing process is outlined in Figure 2.8.

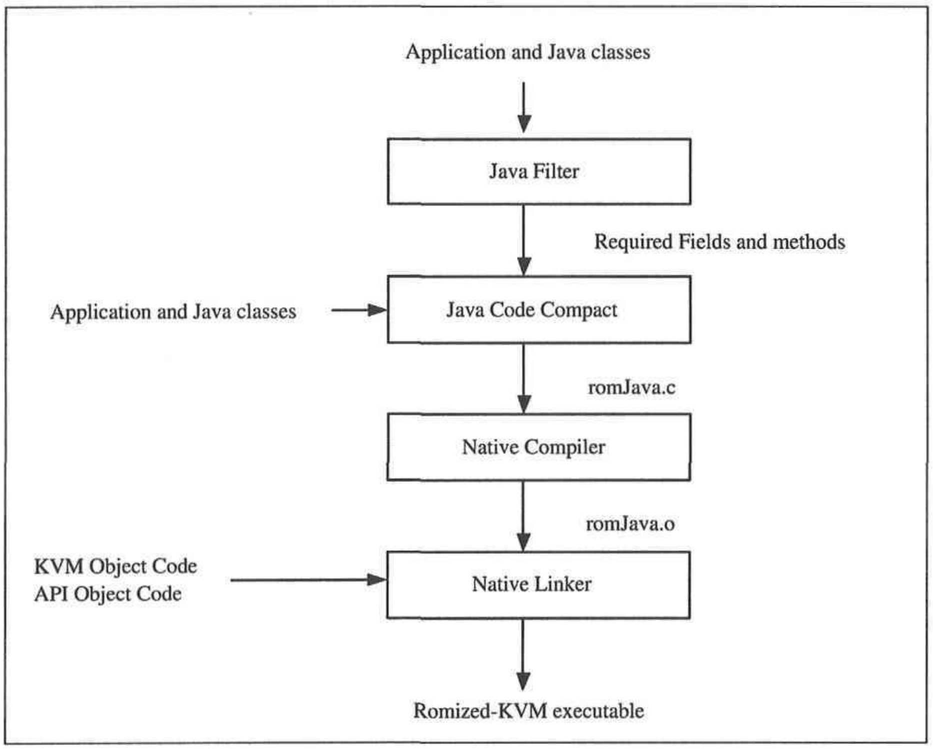


Figure 2.8: Romizing process

The CLDC specification does not allow user-defined class loaders. The rationale behind this is to prevent security flaws. This decision is one of the global strategy of the sandbox security model of J2ME. In this model, the applications are executed in a restricted environment so they can use only non-critical resources.

Class Verification

The JVM class verification process is not convenient for the KVM. In fact, the data flow analysis performed by the verifier requires a lot of memory. In the KVM, the verification process is dispatched into two steps:

- A pre-verification step: in this step, a data flow analysis is performed. This analysis collects typing information about program variables. The computed information is stored in the pre-verified class file.

- A lightweight verification step: in this step, a verification of the information stored in the pre-verified class file is performed.

The dispatch of the verification into two steps makes the verification fast because the most costly step is performed offline.

Interpretation

The KVM interpretation mechanism is based on two interpretation loops: one slow loop and another fast one called the main loop. Each loop contains a switch-case instruction. The fast loop handles frequently executed bytecodes (e.g. arithmetic, load and store bytecodes). It is made as small as possible to allow compilers to optimize it. The slow loop is called by the main loop when it encounters an infrequent bytecode.

Another optimization introduced in the KVM consists of the use of Quickened forms of bytecodes. These bytecodes allow to get quickly some information about the executed code from a dedicated cache. For instance, if a call receiver class is not changed w.r.t a previous call, the method can be extracted directly from the cache without the need to look for it again in the hierarchy. This situation is related to the bytecode invokevirtual_fast, which is a quickened form of the bytecode invokevirtual.

The KVM interpretation main loop is depicted in Figure 2.9. The interpreter uses the KVM stack to save contexts of calling and called methods. The principle of saving/restoring these contexts is the same as what is presented in section 2.2.2. The KVM stack layout is depicted in Figure 2.10.

The interaction between the KVM interpreter and the native code is performed via some basic primitives that allow the manipulation of KVM and native stacks. Traditionally, a Java Native Interface (JNI) is used in conventional VMs to allow such communication. However, for security and cost reasons, this mechanism is not supported in J2ME.

Threading

Thread Synchronization

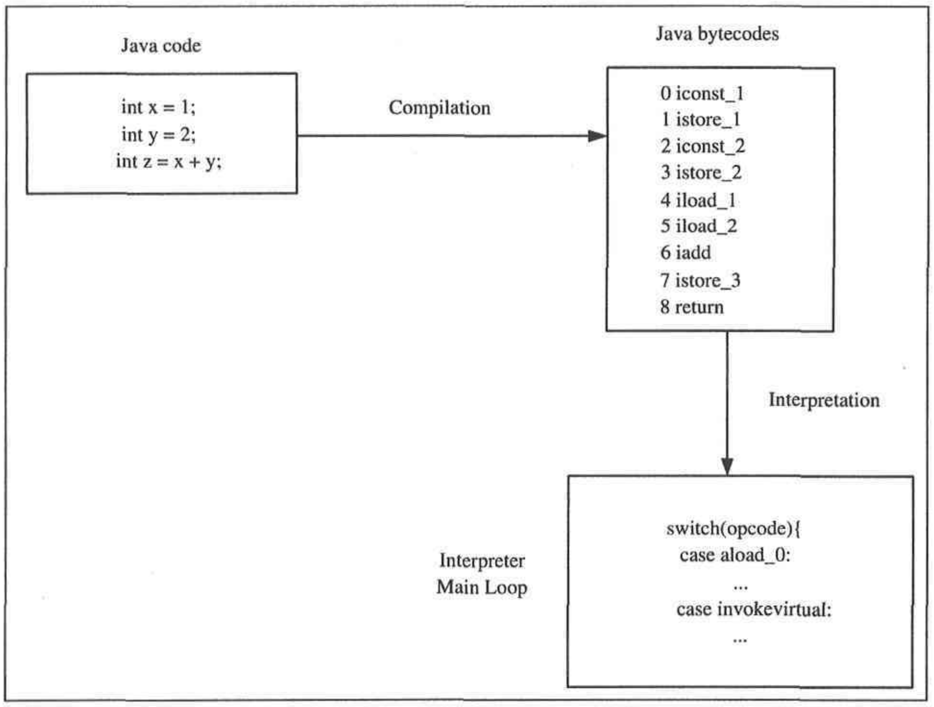


Figure 2.9: Interpreter

The KVM implementation supports threading. However some aspects of threading such as thread group management are not supported. Figure 2.11 shows an automaton that represents the KVM synchronization model. The automaton shows object locking states as circles and the state transitions are shown as directed lines connecting the automaton states. The automaton consists of states *A*-*E*, which indicate the locking status of a given object. The state *A* represents an unlocked state; the state *B* represents a simple locked state; the state *C* represents an extended state; the state *D* represents a monitor state; and the state *E* represents an exception state. The transitions are identified with letters: '*u*', '*s*', '*e*', '*m*' and '*x*'. In particular, the letters represent the following operations: *u*: set_unlocked; *s*: set_simple_lock; *e*: set_extended_lock; *m*: set_monitor; and *x*: raise_exception.

The interaction of the transition operations is described in the context of a given object. Initially the object is in an unlocked state *A*. A set_simple_lock() operation is performed when a thread attempts to lock the object for the first time. The object's locking state changes to the simple lock state *B*. Further, when the same thread attempts to lock the same object, which is in the simple locked state *B*, the object's locking state is changed to the extended lock state *C*. The object remains in an extended state *C* until any different thread tries to lock it further. From any given state, a creation of a monitor state *D* can happen when a second thread tries to lock an object while

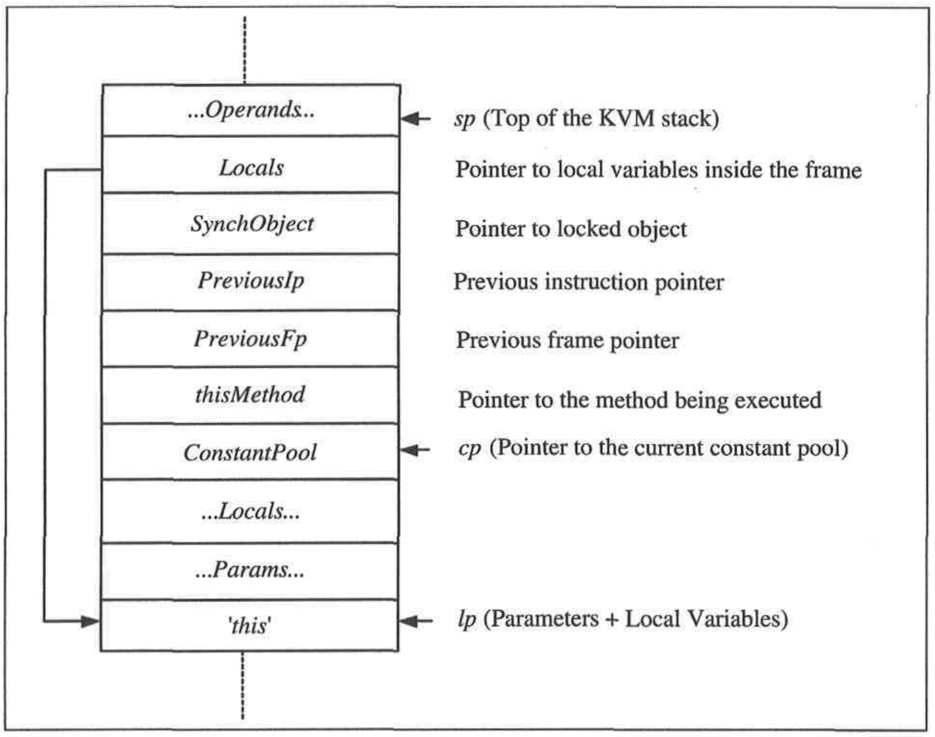

...Operands..
Locals
SynchObject
PreviousIp
PreviousFp
thisMethod
ConstantPool
...Locals...
...Params...
'this'
sp (Top of the KVM stack)
Pointer to local variables inside the frame
Pointer to locked object
Previous instruction pointer
Previous frame pointer
Pointer to the method being executed
cp (Pointer to the current constant pool)
lp (Parameters + Local Variables)

Figure 2.10: KVM stack layout

another thread owns the lock. In this case, a transition from any state to the monitor D state happens. Exiting from a synchronized method triggers the transition from the monitor state D or the extended state C to the state A.

A transition from any given state to any other state is possible with the sole exclusion of the exception state E. An object reaches the exception state E when an exception signal is raised. For all other states, i.e., A-D, transition to any other state or to itself is possible by sending an appropriate transition instruction or signal. The state A is final since in the case of a normal execution (without exception), any object should be unlocked.

Thread Switching

The KVM manages thread execution by switching between them. In fact, after executing a certain number of bytecodes, thread switching can be performed. The execution environment of the current thread is stored and that of the new thread is loaded. This rescheduling mechanism is based on a round-robin algorithm. Notice, that for portability goals, the multi-threading management is platform-independent.

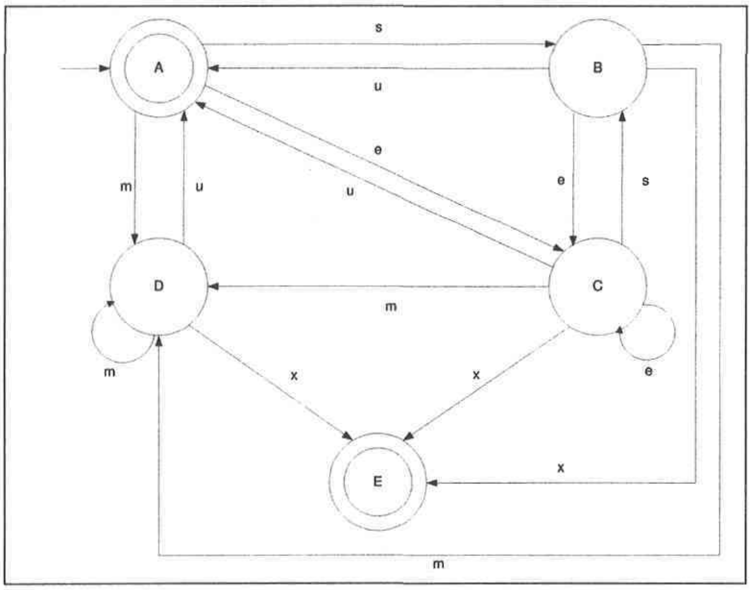


Figure 2.11: Thread synchronization in KVM

Garbage Collection

The garbage collection eliminates unused references in the heap. This helps in avoiding memory overflow errors. In the JVM, a finalization process is responsible for freeing memory. However, there is no support for the finalization process in the KVM because it consumes time, which is unacceptable for limited configuration devices.

Garbage collection in the KVM is implemented through a mark and sweep with compaction algorithm. This algorithm operates in two steps:

1. The first step consists of visiting the heap and marking all alive objects. A bit is used to mark each alive object.
2. The second step consists of the removal of unmarked objects from the heap. These objects are considered unused. However this can create a memory fragmentation problem. A compaction of the heap allows to avoid this.

2.3.2 Mobile Information Device Profile (MIDP)

Owing to the wide range of configurations of wireless devices, Sun has introduced the concept of a Profile to the J2ME platform in order to address this issue. A profile provides the libraries to develop applications for a particular type of devices. Specifically, the Mobile Information Device Profile (MIDP) introduces a set of J2ME APIs that define how software applications interface with cellular phones, pagers, PDAs, etc., taking into consideration the screen and memory limitations of these wireless devices. More specifically, MIDP offers APIs that enable developers to write wireless applications that use user interface components, I/O and event handling, networking, etc. Interfaces are built by using/extending a very specific class called *Midlet*. This class is the equivalent of a Java applet in desktop and server systems.

2.4 KVM Reverse Engineering

Our first mission was to understand and extract the relevant functionalities in this VM. This is not possible with only reading 24,000 lines of C code. Accordingly, we used static and dynamic analysis tools to investigate the inner working of the KVM. In the sequel, we present the static and dynamic analysis of the KVM and also the benchmarking of this VM on an embedded device.

2.4.1 Static Analysis of the KVM

By static analysis, we mean the static investigation of KVM components. We achieved this by the use of an interesting tool called *Imagix* [28]. This tool allows the reverse engineering of C and C++ code. Moreover, it provides a graphical representation of the call graph of the analyzed code. Hence, the user can navigate through call links and discover the possible interactions between the components of an application.

Figure 2.12 presents a snapshot of the Imagix analysis output for the KVM.

Figure 2.14 outlines a partial view of the KVM call graph extracted by Imagix. The call graph shows that the loading and initialization of the main class are performed first and after that the interpretation starts. Notice that the interpreter can interact with the loading and initialization components. This happens when a class needs to be loaded or initialized.

2.4.2 Tuning KVM Performance

The intent of the dynamic analysis of the KVM is to know the frequently called components. These are qualified as critical ones. Such investigation, helps in focussing our efforts on the relevant components that should be enhanced.

VTune [1] is a very interesting tool for performing this kind of analysis. By a dynamic profiling of the KVM, VTune can provide statistics about the time cost of each function call. A dynamic call graph, where critical functions are colored in red, is also provided. Figure 2.15 outlines the KVM tuning.

By this analysis, we found that the *interpret* function is the most critical one. So, the acceleration of the interpretation is worth to be done. This conclusion comes after performing the analysis on many applications and specifically on a standard benchmark called *CaffeineMark* benchmark. The *CaffeineMark* benchmark provides five tests:

- Sieve: computes prime numbers < 2048.
- Loop: executes many loops.
- Logic: executes logical operations (e.g. shift).
- String: executes many string concatenations.
- Method: executes recursive calls of methods.
- Float: simulates a 3D rotation of objects around a point.

For each test, a score is computed. A higher score indicates better performance. An overall score, which is a function of individual test scores, is provided in order to allow the comparison of several VM performance. This standard benchmark is used later to evaluate our acceleration techniques.

2.4.3 Benchmarking the KVM on Embedded Platforms

In order to evaluate KVM performance in embedded devices, we ported the original KVM to ARM[1]. We had extended the KVM to support floats. The main goal is to know KVM performance on these devices. This is done by executing *CaffeineMark* on this ported KVM in a Compaq IPAQ PDA that is running Linux. Figure 2.13 outlines KVM performance on this device. As shown, the floating score is very low. This explains why floats are not supported by the first CLDC specification (version 1.0).

2.5 Conclusion

In this chapter, we provided a quick overview of Java fundamental concepts. We presented the embedded Java platform (J2ME), which is designed for resource-constrained devices. Particularly, we gave some technical details about the design of the KVM and its main components. The next chapter is devoted to the related work about Java acceleration techniques.

[1]ARM is the architecture of most of the embedded devices

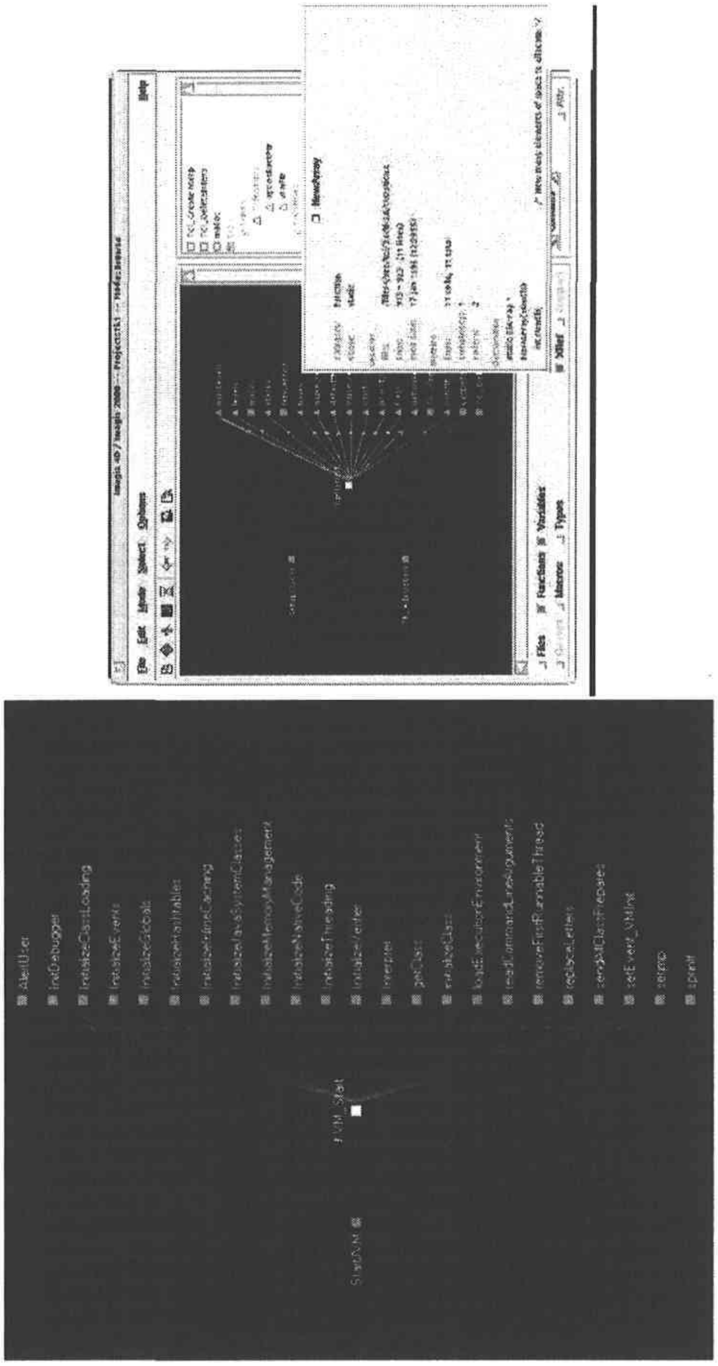


Figure 2.12: Snapshot of KVM analysis in Imagix

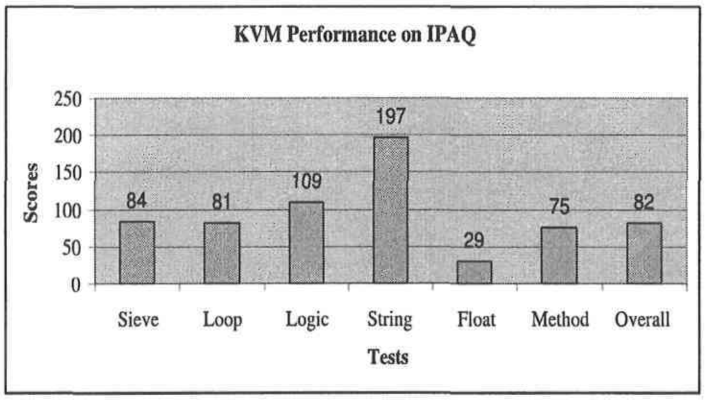


Figure 2.13: KVM scores on an IPAQ

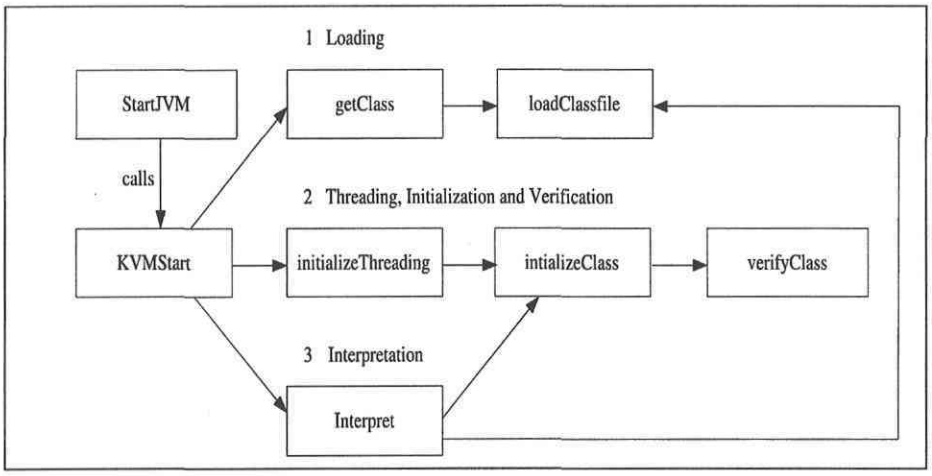
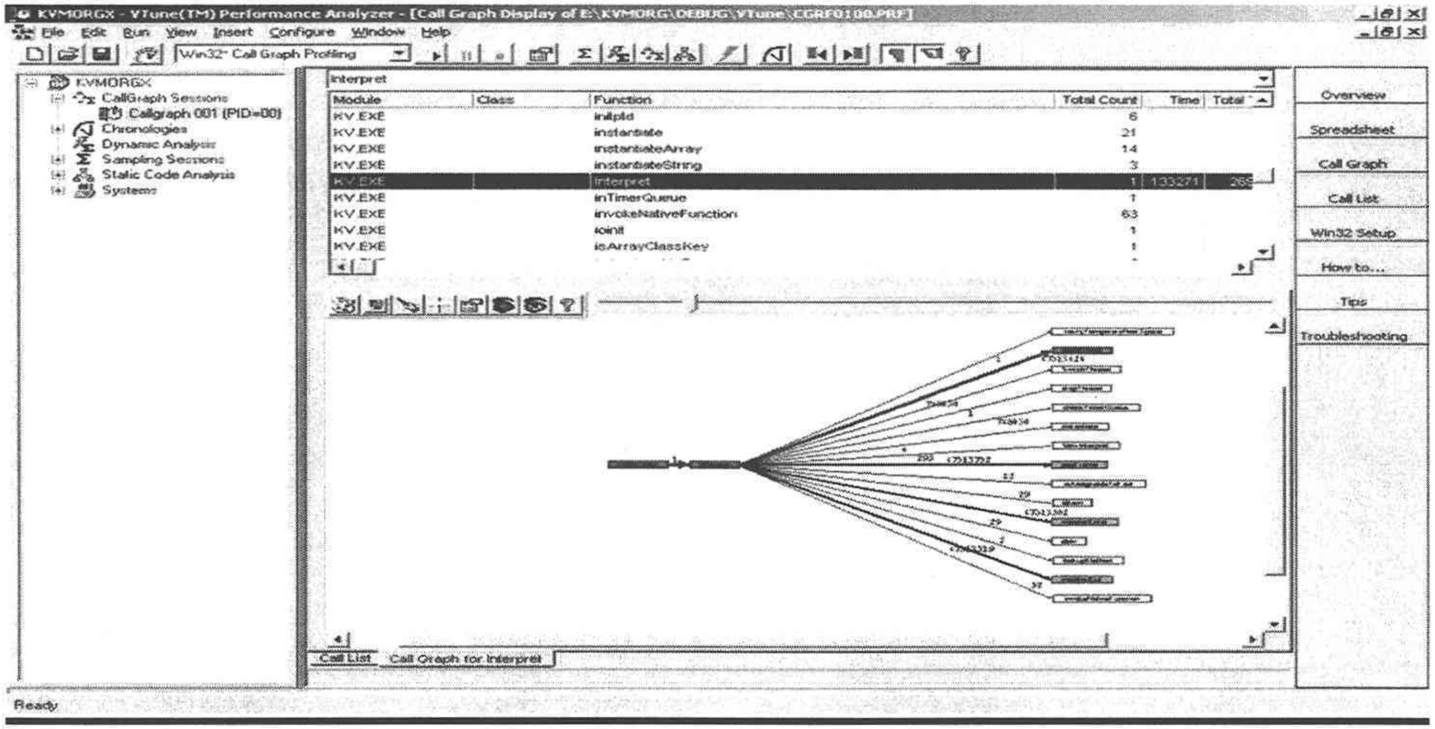


Figure 2.14: Partial view of the KVM call graph



KVMORGX - VTune(TM) Performance Analyzer - [Call Graph Display of E:\KVMORG\DEBUG\VTune\CGRF0100.PRF]

Interpret

Module	Class	Function	Total Count	Time	Total
KV.EXE		initpid	6		
KV.EXE		instantiate	21		
KV.EXE		instantiateArray	14		
KV.EXE		instantiateString	3		
KV.EXE		Interpret	1	133271	265
KV.EXE		inTimerQueue	1		
KV.EXE		invokeNativeFunction	63		
KV.EXE		ioinit	1		
KV.EXE		isArrayClassKey	1		

Figure 2.15: KVM tuning

Chapter 3

State of the Art of Java Acceleration Techniques

3.1 Introduction

The intent of this chapter are twofold: first present a taxonomy of Java acceleration techniques and second exhibit the issues facing the acceleration of embedded Java platforms.

Particularly, we give insight into the dynamic compilation as a relevant acceleration technique for embedded Java platforms. Then, we detail a plethora of optimization techniques implemented by dynamic compilers (called also Just In Time compilers) and highlight their strengths and weaknesses. finally, we exhibit dynamic compilation challenges in the embedded context.

It is worth to mention that the related work about concurrency models is detailed in chapter 6.

3.2 Taxonomy of Acceleration Techniques

There are two main techniques for accelerating Java Virtual Machines: hardware and software acceleration.

3.2.1 Hardware Acceleration Techniques

In the sequel, we provide the available information about hardware acceleration of Java. These information are not very detailed and are generally available in web sites of companies or white papers. This is due mainly to the commercial goals of the acceleration projects.

JSTAR Interpreter

The JSTAR Nazomi's interpreter [27] is an on-the-fly interpretation engine that generates native code from bytecodes. JSTAR supports 159 bytecodes directly in hardware. JSTAR sit in the instruction decode path and translate Java bytecodes, on the fly, into the native code of the target host. JSTAR's interpreter takes over the main interpretation loop and, where appropriate, passes control back to the VM. To allow this, Nazomi rewrites the VM by introducing a call back table. When JSTAR encounters a Java bytecode that does not belong to the 159 supported bytecodes, it performs a call back to a VM function. This is generally dedicated to complex bytecodes like `invokevirtual` since these bytecodes are complex and consume a lot of energy if they are executed directly in hardware. JSTAR's performance is validated using the industry standard Embedded CaffeineMark benchmark [118]. Individual Embedded CaffeineMark tests run up to 10 times faster with JSTAR than on the standard KVM.

Jazelle Interpreter

The ARM Jazelle interpreter [7] is tied to ARM architecture. This makes it different from JSTAR, which is able to work with any CPU. Like JSTAR, Jazelle translates bytecodes, on the fly, into native code. In Jazelle, 140 bytecodes are supported directly in hardware. In addition, two instructions are used to switch between the native state and the Java state. Jazelle technology removes the interpretation loop from the VM and replaces it with ARM's propriety support code. According to ARM, Jazelle increases Java application performance by a factor of 8 compared to an equivalently ARM processor executing Java.

JVXtreme Hardware Accelerator

The Insilicon JVXtreme [63] is a Java processor that decodes bytecodes and executes them as well. JVXtreme handles 92 bytecodes directly in hardware and, like Jazelle and JSTAR, it leaves the other bytecodes (more complex ones) for software execution on the host CPU. When JVXtreme encounters an unsupported bytecode, it calculates a jump address and passes control back to the host CPU. JVXtreme has an instruction-folding mechanism that looks for situations in which several bytecodes can be executed in the same cycle. JVXtreme performance is estimated to 15 times over a Java software interpreter.

Chicory Systems HotShot Engine

The Chicory Systems HotShot engine [24] implements a hardware-based Just In Time compiler (JIT). Like the other three implementations, Chicory HotShot handles a subset of the Java bytecode (148 bytecodes) directly in hardware, leaving the more complex bytecodes for software execution on the host CPU. The HotShot architecture supports "Quickened bytecodes". These bytecodes allow a fast execution since they capture information that allow to avoid reference resolution. Chicory HotShot increases the execution speed of Java applications by a factor of 25 times on embedded devices.

Zucotto's Xpresso Processor

Zucotto Wireless [91] has developed a Java processor called Xpresso. Like the other hardware accelerators, Xpresso supports only a subset of bytecodes directly in hardware. When the Xpresso processor encounters one of the unsupported 56 bytecodes, it yields control to a software module, which executes the Java instruction. To allow direct memory access and to improve virtual machine performance, Xpresso implements some custom bytecodes. Moreover, Zucotto includes an efficient hardware support for garbage collection. It is worth to mention that Zucotto put a high priority on power consumption. Xpresso delivers 20 to 40 times of execution speedup w.r.t the KVM.

In what we presented above, it is clear that hardware accelerators achieve a significant speedup of Java execution. However, it remains that their use comes with a high price in terms of power consumption. This energy issue is really damaging especially in the case of low end mobile devices. Moreover, the cost (royalties, licensing, etc.) of these hardware acceleration technologies is an additional obstacle to their wide adoption by the industry.

These drawbacks of hardware acceleration created an interesting and challenging niche for software acceleration of embedded Java virtual machines. Software acceleration techniques are classified in three categories: ahead of time compilation techniques, general optimizations and dynamic compilation. These techniques are detailed in what follows.

3.2.2 Ahead of Time Compilation (AOT)

Ahead-of-Time compilation is a compilation technique that is applied before the execution. This allows to perform traditional time-consuming optimizations such as intraprocedural/interprocedural data-flow analyses and optimizations. Consequently, the generated code is of high quality. AOT compilers are classified in two subcategories: translators and direct compilers.

Translators

Translators perform the transformation of application code into an intermediate high level programming language. Because C compilers allow several optimization techniques and are available for almost all the platforms, many bytecode-to-source compilers use C as an intermediate high level language into which Java bytecodes are translated. An example of such translators is Toba, which speeds up Java execution by a factor between 1.5 and 4.2 [106]. Toba has its own object layout and its own threading, exception and garbage collection mechanisms. The main drawback of Toba is that it does not support the interaction with the Java interface since all the classes, including APIs, are translated into the C language. Moreover, the translation of all Java classes induce a high memory footprint, which is not acceptable in embedded devices. Another drawback consists of supposing that all the classes are available at execution, which is not always true because a Java application can load at execution time a class from a distant website.

Direct Compilers

Direct compilers translate application code directly to native code. TurboJ [134] is an example of a direct compiler for Java. This compiler does not generate a standalone binary code. Instead, it generates native code that will interface with a Java Virtual Machine. TurboJ selects some classes for compilation in order to reduce the size of the generated code. By doing so, the required memory space for storing this code is reasonable.

TurboJ performs offline optimizations before generating a binary code. These optimizations include transforming method calls to direct calls when the call receiver is known and replacing a method call by its code (inlining). The results show that TurboJ speeds up Java execution by a factor of more than 10 times.

As aforementioned, there are three main disadvantages of AOT compilers:

1. The memory footprint entailed by these compilers is important even if a selective compilation technique is used. In fact, by compiling an entire class, some of the compiled methods are not really invoked. Hence, their compilation induces a time overhead.
2. They can not optimize dynamically-loaded classes since the compilation is performed offline. Moreover, in an offline mode, the application source code is not always provided so it is not reasonable to suppose that the code (or part of it) is available before the execution. Nevertheless, this technique can be relevant for Java APIs.
3. The generated executable code is not always portable. In fact, the platform is not known in advance. Alternatively, a dynamic compiler has the advantage to be able to detect dynamically the target platform and generate a convenient code for this platform.

3.2.3 Dynamic Compilation

The dynamic compilation technique consists of the dynamic translation of a source code into native code. For Java, this technique can improve considerably the execution time. In fact, the stack-based representation of Java bytecodes relies on many stack manipulations. The values of the variables are loaded from/into the stack frequently.

These operations make the interpreter slow. On the other hand, the native code is register-based. Consequently, the translation of Java bytecodes to native code speeds up the execution of the translated code.

JIT compilers try to get the best tradeoff between code quality and acceleration. In fact, getting a code that uses the minimum number of machine registers needs performing variable liveness analyses. These analyses are expensive. Avoiding these analyses offers a fast compilation but provides a native code that use many memory load and store operations (called also a spill code). To give an idea about JIT compilation, we present in the sequel some important definitions.

Intermediate Representation

Most of JIT compilers generate an Intermediate Representation (IR) from the application code. For Java, the bytecode representation is not convenient for performing data flow analyses and optimizations. An IR like a three-code address representation is easier to analyze.

Register Allocation

JIT compilers face a problem that traditional compilers must solve. This problem is the register allocation, which is performed at the code translation step. Register allocation is the process of allocating machine registers (or physical registers) to variables. Since the number of machine registers is finite, the problem of allocating the minimum number of registers to the maximum number of variables is similar to a graph coloring problem [23].

Dynamic Compilation Techniques

There are two known dynamic compilation techniques: a traditional JIT compilation [40] and a dynamic selective compilation technique [78]. In the traditional JIT compilation technique, a method code is translated into native code at the first invocation of this method. This technique could increase considerably the memory overhead when the application code to be compiled is huge. Moreover, a compilation time overhead is introduced when useless code is compiled.

The selective dynamic compilation technique avoids the compilation of rarely invoked methods by selecting the methods that are frequently called. This technique had proven its relevance because it is time and memory-space aware. The main important components of a dynamic selective compiler are: a profiler, a compiler and a cache manager. In what follows, we present the role of each component.

Profiler

The main profiler task is to predict on which pieces of code the program spends most of its time. For this end, the profiler must be able to monitor and trace events that occur during runtime, to set the cost of these events and attribute the cost of these events to specific parts of the program [71].

Hence, the profiler uses the past to predict the future and, the more prediction is made early, the more the JIT compiler is efficient. Indeed, if the profiler is too timid in its decision making, it may miss good opportunities for optimization. There are three categories of profilers: time-based, counter-based and sample-based.

- Time-based profiler

  A time-based profiler records the time spent in each method. Instrumentation instructions are inserted in calls, returns, throws and catches. These instructions check the current time and add it to the appropriate method. Once, a method reaches a pre-established time threshold, it is considered to be frequently called (known as *hotspot*). The advantage of time-based profiling is its completeness in the sense that all methods appear in the profile. However, the major disadvantage, is the overhead introduced in both, time and code size, by instrumenting instructions. This disadvantage reduces considerably the use of time-based profiling in dynamic compilation systems.

- Counter-based profiler

  A counter-based profiler allocates an invocation counter to each method. At the beginning of the execution, all invocation counters are initialized to zero. Whenever a method is invoked, the corresponding counter is incremented. The invocation counter is incremented on method entries and backward branches (loop iterations). The advantage of counter-based profiling is its completeness and accuracy. The disadvantage is the overhead caused by counters update and the issue of choosing thresholds.

- Sample-based profiler

  This profiler samples the running application periodically (every 10 milliseconds in several systems), when the application reaches predefined points. These points are method entries and loop back-edges. A possible strategy to attribute samples to methods is the following:

  - When the sample is taken on a loop back-edge, it is attributed to the method containing the loop,
  - When it is taken on a method entry, it is attributed to both the calling and the called method.

  Another strategy to sample the running application consists of traversing the Java stack periodically in order to see which methods are currently executed. Samples are attributed to these methods. Then, methods are ordered according to the number of samples attributed to them. Once, the number of samples exceeds a pre-established threshold, the method is considered to be frequently called. An important advantage of this type of profiling is the reduced overhead. Indeed, there is no need to instrument the code at each method entry or loop back-edge and sampling can be triggered periodically. However, the disadvantage of sample-based profiling is its incompleteness since it is possible for a method to be executed but never being sampled.

Compiler

The compiler is responsible for generating native code from Java bytecodes. The compilation unit can be a method or a fragment of code. If the compilation unit is a method, then the cache can be quickly saturated. On the other hand, if the compilation unit is a piece of code, then the cache capacity can be better exploited but the detection of frequently used fragments is more complex. Since memory space is limited, the generated code is saved in the cache for future calls.

Cache Manager

A cache structure is needed for storing the compiled code. The compiled code typically resides in the heap. So, it can be removed by the garbage collection like any other object. The cache manager is responsible for managing the native code in the memory. Since the cache size is limited, the cache manager must select some elements to remove. This selection depends on many parameters such as method size, compilation time and future calls of this method. Consequently, a cache replacement policy is required. In the sequel, we present some known algorithm for memory page replacement in operating systems. These algorithms can be applied to cache management for Just-In-Time compilers.

1. First-In-First-Out (FIFO) Algorithm: it takes into account the chronological order of the element storing. Thus, when the cache is full, the element to be removed is the oldest among those currently in the cache.
2. Optimal Algorithm: it replaces the element that will not be used for the longest period of time. This algorithm is used mainly for comparison studies. In particular, it is used as benchmark in order to compare other algorithms. In fact, this algorithm is not achievable in practice because it requires a perfect knowledge of the future (the entire list of elements that will be referenced).
3. Least-Recently-Used (LRU) algorithm: it takes into account the chronological order of the use of the elements. In fact, it replaces the element that has not been used for the longest period of time. It is an approximation of the optimal replacement algorithm since it uses the past behavior as a predictor of the future. A time of the last use is associated to each element. Hence, this information has to be updated each time an element is referenced.
4. Second chance Algorithm: this algorithm is an enhancement of the FIFO algorithm. It uses a FIFO queue. A reference bit is set whenever the element is referenced. To replace an element when a cache miss occurs, the FIFO algorithm is run but the reference bit of the oldest element is inspected. If it is 1, the element is put at the bottom of the list and the reference bit is reset. Hence, it is given a second chance. Else, if it is 0, the element is old and not used, so it is replaced. The goal of the second chance algorithm is to keep recently used elements in the cache.

3.2.4 General Optimizations

General optimizations aim to accelerate several JVM features such as threading, method lookup, garbage collection and interpretation. In the sequel, we present the motivation behind accelerating each feature and how it can be accelerated.

Threads Synchronization

Java threads synchronization mechanism leads to a potential performance overhead. In fact, Java libraries are implemented in a thread-safe manner (by lock and unlock primitives), so the synchronization mechanism degrades considerably the application performance, particularly for mono-threaded applications. In this case, the synchronization mechanism is not appropriate because there is only one running thread. In Java, the synchronization mechanism is provided through monitors that give exclusive access to shared data. This structure contains some information about object lock states that have to be stored in the object header. Hence, an additional per-object state (for lock and unlock operations) and at least two words (sometimes three) of object header space are needed.

A known optimization technique [33] called *Thin Locks* implements monitors in a lightweight manner. A minimum set of instructions is used to perform lock operations on objects. Besides, a compact memory representation of a monitor allows to minimize the memory footprint. This strategy allows to accelerate the acquisition and the liberation of the lock.

Other research initiatives [12] showed that object header compression techniques can improve run-time performance. Particularly, a good object model design is required to speed up the synchronization mechanism.

However, the main disadvantage of these techniques is that they require the change of the object layout, introducing hence changes in other virtual machine components.

Method Lookup Acceleration

Java, like any object oriented-language, uses a dispatch mechanism to implement method calls. In this context, the selection of the appropriate method to execute is based on a lookup mechanism. In fact, the actual method to be executed after an invocation is determined dynamically. This lookup is based on the type of the call receiver, the class hierarchy and the method inheritance or overloading schema. If the receiver class implements a method that has the same signature as the called one, the found method will be executed. Otherwise, the parent classes will be checked recursively until the searched method is found. If no method is found, an error is signaled (*MsgNotUnderstood*). This algorithm is called the Dispatch Table Search [54] (DTS). It induces an expensive cost while probing classes that did not implement a called method. This overhead is up to 29% of the total execution time of an application [21]. Hence, there is a need to speedup up the lookup mechanism. Accordingly, static and dynamic techniques were proposed to minimize the method invocation overhead.

Selector Table Indexing (STI) is a static technique for the acceleration of the method lookup mechanism. It works as follows: given a class hierarchy of C classes and S method names (called selectors), a two-dimensional array of $C*S$ entries is built. Classes and selectors are given consecutive numbers on each axis and the array is filled by pre-computing the lookup for each class and selector. An array entry contains a reference to the corresponding method or to an error routine. These tables are computed for the complete application. The main drawback of STI is that space requirements are huge for a big application. Hence, many dispatch table compression techniques were proposed (Selector coloring, Row displacement, etc.) to minimize space waste. Moreover, this technique is very sensitive to changes in the class hierarchy. Changes in the hierarchy makes the previously computed table inaccurate.

Dynamic techniques consist of caching results of previous lookups, thus avoiding other lookup operations. The main approaches to caching are: global caches, small inline caches and polymorphic inline cache.

The global cache technique stores previous lookup results. In the global cache table, each entry consists of triplets (receiver class, selector, method address). The receiver class and the selector are used to compute an index in the cache. If the current class and the method name match those in the cached entry at the computed index, the code at the method address is executed. Hence, unnecessary method lookup is avoided. Otherwise, a default dispatching technique is used and at the end of this search, a new triplet is added to the cache table and control is transferred to the found method.

The inline cache technique consists of caching the result of a previous lookup (method address) in the code itself at each call site. Inline cache changes the call instruction by overwriting it with a direct invocation of the method found by the default method lookup.

Polymorphic inline cache is an extension of the inline cache technique. The compiler generates a call to a special stub routine. Each call site jumps to a specific stub function. The function is initially a call to a method lookup. Each time the method lookup is called, the stub function is extended.

The main drawback of caching techniques is that a frequent changes of the call receivers induce frequent update of the cache.

Garbage Collection

The garbage collection mechanism manages allocation and deletion of objects. There are two main categories of garbage collection algorithms: reference counting and tracing.

Reference counting algorithms use counters that are associated to memory cells. A memory cell counter is incremented when the memory cell is used and decremented otherwise. When the counter reaches zero, the memory cell is freed and can be allocated to new created structures.

Two known tracing algorithms are deployed in JVMs:

- Mark and sweep algorithms: garbage collection is performed in two steps. The first step is for marking alive objects. In the second one, unused objects are removed from the memory. Then a possible compaction of the heap can be performed.
- Generation-based algorithms: in these algorithms, mainly two generations are used: old and new ones. The old generation contains the objects before allocation and the new generation contains just those newly created. A copy of alive objects from the old to the new is performed. Remaining objects in the old generation are removed. The new generation becomes the old one for further allocations.

In embedded platforms, memory space is limited. This makes the use of generational-based algorithms inadequate. Mark and sweep algorithms use just a bit to mark each object while counting algorithms use an integer. This makes the mark and sweep algorithm more convenient for embedded platforms. Notice that this is the algorithm that is implemented in the KVM.

Interpreter Acceleration

Java applications take more than 60% of their time in interpretation. It is clear that without a fast interpreter, the execution is slow. Accelerating the interpretation mechanism has been and is still a focus of interest for many researchers.

Generally a pure bytecode interpreter is a loop embedding a switch-case statement that dispatches to a sequence of bytecodes. Each switch case value implements one Java bytecode. This entails a significant overhead. To circumvent this drawback the use of direct threaded interpretation has been suggested. The latter is is an interpretation technique introduced in the Forth programming language [45]. Thanks to this technique, the central dispatch is eliminated. Each bytecode of the method being interpreted is replaced by an address of a corresponding implementation. In addition, such an implementation ends with the required dispatch to the next opcode.

The inline threading interpretation technique [101] improves upon the direct threading technique by eliminating the dispatch overhead within basic blocks. The former technique identifies bytecode sequences that form basic blocks. A new implementation is then dynamically created for such sequences by copying and catenating each bytecode's implementation in a new buffer. The dispatch code is then copied at the end. More recently, a 'preparation sequence" has been proposed [48] to cope with the difficulties that arise when adapting the inline threading technique to Java. In fact, Java's features such as lazy class initialization, lazy class loading and linking and multi-threading create a conflict with the implementation of inline-threaded interpretation technique.

General optimization techniques can not compete with compilation techniques at the performance level. This makes them a valuable secondary solution that can work together with compilation techniques.

3.3 Accelerated Embedded JVMs

The deployment of Java on embedded systems faces different issues due to small devices features. Among these features, we mention:

- Memory: it is a precious resource on embedded systems. Indeed, cell phones, PDAs and several embedded devices have an available memory less than 512 KB to handle the entire Java runtime environment. In fact, the minimum memory requirements of a small device are: 300 KB of RAM and about 1 MB of flash and ROM. Specifically, in this context, the available memory size is required to store application data (heap), subroutines and return information (stack), loaded classes and possibly buffers of dynamic compilation. Hence, because of severe memory resource-constraints, the memory footprint for the virtual machine and its libraries have to be minimized.

- Battery life: since battery life affects the extent and duration of mobility, battery efficiency should be enhanced. This can be achieved by a fast startup and execution time. Consequently, when the execution is faster, a little power is consumed.

To deploy the Java technology on embedded systems, it is crucial to have Java acceleration techniques that suite small devices requirements. In this context, many companies are working to boost Java embedded virtual machines performance and to reduce memory footprint. Unfortunately, what is published is very limited. All what we present in the sequel is a summary of available white papers or few publications.

3.3.1 Kilo JIT (KJIT)

Sun has produced an accelerated version of the KVM that integrates a small effective JIT, called KJIT [115]. In KJIT, the compilation process is restricted to two passes in order to get an effective compilation: a pre-compilation transformation of a subset of bytecodes then a dynamic compilation of this subset. In the first pass, stack-based bytecodes are transformed into a simple three-address intermediate representation. This representation uses new created local variables in order to avoid creating temporary variables on the Java stack. By doing so, the switching between the Java stack and the native stack is avoided. The dynamic compilation maps each local variable to a register. To make this possible, the number of local variables is supposed to be less than available registers. Another key feature that makes this KJIT effective consists of having a simple and quick cache management.

However, the first-pass transformation induces an important increase of local variables number and of the code size (estimated to 30%). Moreover, the proposed solution is not complete since exception handling, threads rescheduling and garbage collection are handled at the VM side and there is no consideration of possible exception in the generated native code for instance.

The KJIT implementation, based on the KVM version 1.0.2, requires only 60 KB on ARM architecture and it speeds up KVM execution by a factor ranging between 5.7 and 10.7.

3.3.2 Sun CLDC HotSpot (Monty)

The Sun CLDC HotSpot [84], called Monty, is a high-performance Java Virtual Machine for the CLDC configuration. Its architecture and implementation are largely extracted from the HotSpot implementation for J2SE [78] and adapted to the J2ME platform. It is designed to achieve a fast bytecode execution while maintaining a very small device footprint and minimizing the battery consumption. Mainly, Monty uses an optimized bytecode interpreter and a dynamic selective compiler that translates frequently executed methods into native code. It also uses acceleration techniques dealing with fast synchronization, compact object layout, good cache behavior (configurable code cache size), compilation and de-compilation policies and generational garbage collection. Monty is said to be 10 times faster than the KVM.

3.3.3 Insignia Embedded Virtual Machine (EVM)

The Insignia Embedded Virtual Machine (EVM) [62] is a fast configurable and tunable Java Virtual Machine for its Jeode platform: an accelerated platform designed to meet the Java specifications for resource-constrained devices. It delivers an optimal balance of performance with a small memory footprint. More specifically, EVM uses a dynamic selective compiler, a concurrent garbage collection and a predictable system behavior that enables developers to optimize memory requirements without compromising performance. In fact, the Jeode platform contains an event monitor and a memory use analyzer that enable developers to configure and tune the EVM for any kind of embedded devices. While the application is running, the EVM determines which code fragments are the most frequently executed, compiles them and stores the generated code into a memory buffer, whose size is configurable. The EVM may recycle the buffer to optimize performance in the pre-fixed footprint. EVM is said to execute Java applications, on average, six times faster than interpreted virtual machines with a minimal memory overhead.

3.3.4 Esmertec Jbed Micro Edition CLDC (Jbed)

The Esmertec Jbed Java Virtual Machine [113] is developed for real-time systems. It is a Sun certified JVM for small mobile devices. Jbed provides a complete J2ME runtime environment. Jbed compiles, in an offline step, all the application code to native code instead of interpreting it. Also, the Jbed's virtual machine includes a dynamic compilation component called FastBCC (Fast Byte Code Compiler), which compiles on the fly the loaded classfiles and links them into the application.

3.3.5 Acunia Embedded Virtual Machine (Wonka)

The Acunia Wonka virtual machine [20] is a high-performance virtual machine implementation that runs Java applications which are designed for resource-constrained embedded systems. It does not require a host operating system unlike other embedded virtual machines. Its main features are: concurrent garbage collection that enables efficient use of resources, high availability and effective memory usage by minimizing fragmentation. Acunia developers are working on the development of a dynamic compiler called J-Spot.

The study of embedded VMs shows that there is a little published knowledge about their inner workings. The main reason is that these VMs are designed for commercial aims. Hence, a careful investigation of accelerated conventional VMs and mainly of dynamic compilers for desktop or server class systems is required. The goal of this study is to know what are the inner working of dynamic compilers and to study their relevance for embedded platforms.

3.4 Accelerated Conventional JVMs

In this section, we give an overview of the most famous dynamic compilers for conventional VMs that are published so far.

3.4.1 Sun HotSpot

The Sun HotSpot is a JVM [78] that includes two JIT compilers: a server and a client JIT compiler. The first targets server and desktop systems while the latter is designed for more resource constrained systems. The HotSpot client and server compiler use a dynamic selective technique that compiles the frequently called methods on the fly. To determine frequently called methods, a counter is associated with each method. For each method call, its counter is incremented. When the counter exceeds a preset threshold, the method is considered to be hotspot and a compilation request is performed. A thread compiler is responsible for translating a hotspot method. This selective technique creates a *mixed mode* execution. In fact, an interpreted method could call a compiled (translated) one and vice versa.

The HotSpot client compiler generates an IR from Java bytecodes. It translates the instructions of this IR into machine code. The virtual machine runs a Java application and detects the methods that represent "hot" parts of this program. Computing such information can increase the interpretation time. However, these information allow the HotSpot compiler to know what methods to translate. Thus, the time spent to translate rarely called methods is saved. Both HotSpot compilers perform some common optimizations that we describe in the sequel. We note that the server compiler uses more aggressive optimizations than the client.

Method Inlining

Method inlining consists of replacing a dynamic method call by the code of the invoked method. This technique provides good performance. In fact, the method code allows to perform optimizations on the fly. In addition, frame creation overhead is avoided. Method inlining is achieved by using a Class Hierarchy Analysis (CHA) in order to compute an approximation of call receivers. This technique was designed by Dean et al. [34]. It uses the static type of the receiver object and the class hierarchy of a program to compute the possible receiver types of a dynamic call. Let the class hierarchy be that of Figure 3.1. Let *O.m()* be a method call where O is a variable and A is its declared type; so the possible receiver class could be A, B or C. By analyzing the class hierarchy, we detect that the method m is not defined in the class C nor in B. Hence, the only possible receiver class is A.

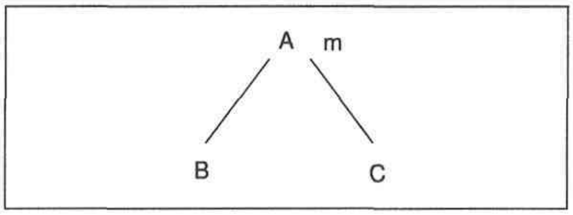


Figure 3.1: Class hierarchy example

However, method inlining could increase memory requirements. Besides, method inlining could give erroneous results in the presence of dynamic class loading.

Dynamic Deoptimization

Java dynamic loading could make an inlining decision erroneous. In fact, the inlined method could not be the same as that invoked when a dynamic class loading occurs. Deoptimization consists of replacing an inlined method with the initial method call. Only the HotSpot JIT server compiler uses this deoptimization technique because it induces an important execution time overhead.

Memory Model

The Sun JVM HotSpot proposes a new object memory model in which the access to objects is performed directly and not through an indirection level. This representation accelerates the access to objects. Moreover, object headers are represented by two machine-words while in ordinary JVMs these headers are represented by three. This optimization enables to reduce memory space requirements. In fact, 8% of space is saved for most of the applications.

A unique stack model is used for Java, compiled and C methods. The goal is to make the communication between Java and the native environment efficient. Besides, Java threads are mapped to native threads in order to profit from the scheduling mechanism used by the OS.

Register Allocation

The first technique that was designed in compilers to perform register allocation is the graph coloring. A graph coloring algorithm operates as follows:

- Build basic blocks of an IR-based program and compute live variables at each instruction.
- Build an interference graph. Each node of this graph is a program variable. An edge links two nodes if they are live at the same time at least at one instruction.
- Coalesce nodes that are sources and targets of copy operations.
- Try to find an N-coloring subgraph of the interference graph where N is the number of machine registers.

The HotSpot server compiler uses a register allocation algorithm designed by Chaitin [23]. The idea of Chaitin is to find a node T of degree less than N. The interference graph is N-colorable if the sub-graph, from which the node T and its edges are removed, is N-colorable. This step is recursively applied until all the graph is traversed. The blocking step occurs when the algorithm can not find a node of degree less than N. In this case, a spill code is generated (a code that copies a register to a temporary variable). The interference graph is modified (by removing the spilled registers). This is repeated until an N-colorable code is obtained. The removed nodes are added in the reverse order by which they were removed. For each added node, a color (a machine register) is assigned.

HotSpot Performance

Before presenting the HotSpot performance, we give an idea about a famous industrial benchmark called *SpecJVM98*. This benchmark is composed of the following programs:

- _200_check: a test that deals with array creation, public and private access, array operations and branching operations.
- _201_compress: runs a compressing algorithm.
- _202_jess: runs rules on a set of data. A puzzle should be resolved. The data set is expanded iteratively by adding a new fact with different literals.
- _209_db: performs some operations on a database.
- _213_javac: runs the javac compiler.
- _222_mpegaudio: decompresses audio files.
- _227_mtrt: this is a raytracer that works on a scene depicting a dinosaur.
- _228_jack: a Java parser generator developed in Purdue University.

For each of the above tests, a score is computed. A test score (called also a spec ratio) is the result of the division of its execution time under a reference machine (the latter is a specific machine that was chosen, see [41] for more details) by the test execution time. A higher score means a better performance. The obtained scores allow developers to evaluate the performance of a JVM or a Java compiler. The overall performance is determined by the geometric mean of unit test scores. In some tables, throughout this document, we give execution times instead of giving the scores. In this case, no geometric mean is presented.

The HotSpot client performance on Solaris/SPARC with 512 MB memory is illustrated in Figure 3.2. Columns represent scores.

Benchmark	Worst	Best	Interpreter
_227_mtrt	40.9	76.4	11.6
_202_jess	25.1	35.2	11.9
_201_compress	45.1	48.4	4.88
_209_db	12.6	14.4	6.59
_222_mpegaudio	44.6	53.6	7.62
_228_jack	21.0	29.6	17
_213_javac	9.36	18.8	8.93
Geometric Mean	24.4	34.3	9.09

Figure 3.2: HotSpot client performance

The results show that the HotSpot client is 3 to 4 times more efficient than a JDK 1.3.1 interpreter. HotSpot client good performance come mainly from the profiling strategy. In fact, only frequently called methods are translated into native code. However, the register allocation algorithm requires a high memory space since it is based on a coloring graph algorithm that uses an interference graph. Building such a graph requires computing liveness information about code variables. Moreover, many steps are needed to translate Java bytecodes into native code. This is inadequate for power and memory-constrained systems.

3.4.2 IBM JIT Compiler

Sauganuma et al. [124] have designed a JIT compiler that performs run-time optimizations for Java applications.

Data Structures Optimization

For instance, data structures are modified in order to accelerate their access. The object layout is modified in order to make the access to it faster. In fact, in the JVM, an

Before overriding the method	After overriding the method
call imm_ca // static method call	jmp dynamic_call
jmp after_call	jmp after_call
dynamic_call:	dynamic_call:
load cp, (obj)	load cp, (obj) // load class pointer
load mp, (cp)	load mp, (cp) // load method pointer
load ca, (mp)	load ca, (mp) // load code address
call ca	call ca
after_call	after_call

Table 3.1: Dynamic call optimization in the presence of dynamic class loading

object is accessed through an indirection level. This indirection level makes the access to object fields very slow.

Inlining

Inlining is one of the performed optimizations. The inlining decisions take into consideration many factors such as the code length of the inlined virtual method, the existence of an exceptions table, the number of local variables and of stack variables increased by the inlining optimization.

Another optimization consists of replacing a dynamic method call with a static one. This is achieved by using CHA [34]. The main obstacle, as mentioned previously, that makes this optimization difficult is the dynamic class loading feature of Java. In fact, the class hierarchy can change dynamically when the virtual machine loads a class that contains new methods that can override some of the inlined ones. The authors had adapted CHA to the dynamic class loading issue by changing a static invalid call with the initial dynamic one. Table 3.1 outlines their solution.

Exception Check Elimination

Some Java dynamic verifications can be avoided in some cases. For instance, a `NullPointerException` check, an array bounds check can be removed by performing some data flow analyses. For the `NullPointerException` check, a flag, saying that the pointer has been tested, is propagated while analyzing the Java code. This flag is tested when a pointer check is required. For array bounds check, a data flow analysis is required to compute the range set of the array indexes and propagate them through the program Control Flow Graph (CFG).

Dynamic Compilation

To avoid a high memory footprint (memory requirements), the proposed JIT compiler performs a selective compilation where hotspot methods are translated into native code. A counter is associated to each method. An initial value is assigned to this counter. This value is considered as a threshold. The counter is decreased by 1 when the method is invoked. A zero value of the counter triggers a translation of the method code into native code. The IBM-JIT native code generation step is performed as follows: a flow analysis is performed on Java bytecodes after generating basic blocks. Then, Bytecodes are converted to an IR called extended bytecodes. An example of an extended opcode is one that obtains a reference to an array. This opcode allows, by store and load operations, to access consecutive array elements instead of indexing each element. After that, some optimizations are performed on the extended bytecodes (method inlining, exception check elimination, common subexpression elimination, etc.). Finally, the code translation step is performed at the same time as the physical register allocation in order to minimize the compilation time. A code scheduling is then performed to get an optimized code representation. Figure 3.3 outlines the architecture of the IBM-JIT compiler.

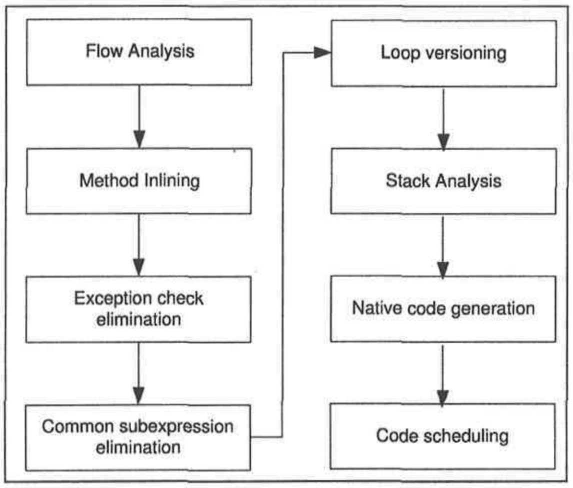
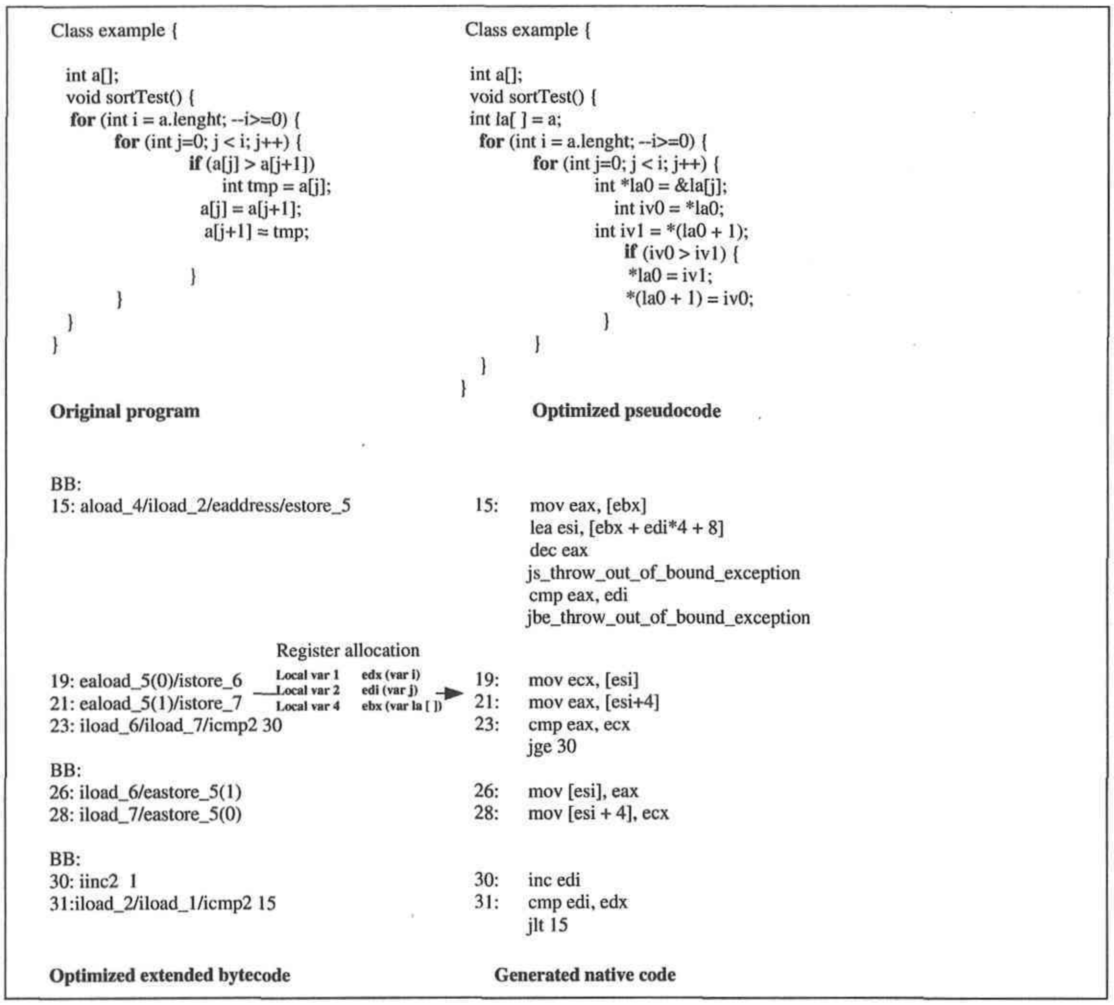


Figure 3.3: IBM-JIT compiler architecture

The IBM-JIT compiler uses its own strategy to generate native code. In fact, the Java code is divided into tiles. A loop is considered as a tile and the code between loops is so. Information about local variables are stored in each tile. Local variables are ordered by their access counting. The register allocation process does not use a graph coloring algorithm because it is expensive. It allocates registers to stack variables first, then to local variables according to their access counting. A circular allocation algorithm that returns the least recently allocated register is used to assign a physical register to a variable. In fact, when a new register is needed and there is no available physical register, a register among those allocated to local variables is chosen and finally among those allocated to stack variables. This order assumes that stack variables are frequently referenced so they are checked last. Figure 3.4 presents an example of a Java program and the native code generated from it by the IBM-JIT compiler.



```
Class example {

  int a[];
  void sortTest() {
   for (int i = a.lenght; --i>=0) {
         for (int j=0; j < i; j++) {
                   if (a[j] > a[j+1])
                        int tmp = a[j];
                      a[j] = a[j+1];
                      a[j+1] = tmp;

                   }
         }
  }
}
```

Original program

```
Class example {

int a[];
void sortTest() {
int la[ ] = a;
 for (int i = a.lenght; --i>=0) {
         for (int j=0; j < i; j++) {
                 int *la0 = &la[j];
                    int iv0 = *la0;
                 int iv1 = *(la0 + 1);
                    if (iv0 > iv1) {
                     *la0 = iv1;
                     *(la0 + 1) = iv0;
                   }
         }
  }
}
```

Optimized pseudocode

```
BB:
15: aload_4/iload_2/eaddress/estore_5

                                        Register allocation
19: eaload_5(0)/istore_6     Local var 1    edx (var i)
21: eaload_5(1)/istore_7    —Local var 2    edi (var j)  →
23: iload_6/iload_7/icmp2 30 Local var 4    ebx (var la [ ])

BB:
26: iload_6/eastore_5(1)
28: iload_7/eastore_5(0)

BB:
30: iinc2 1
31:iload_2/iload_1/icmp2 15
```

Optimized extended bytecode

```
15:   mov eax, [ebx]
      lea esi, [ebx + edi*4 + 8]
      dec eax
      js_throw_out_of_bound_exception
      cmp eax, edi
      jbe_throw_out_of_bound_exception

19:   mov ecx, [esi]
21:   mov eax, [esi+4]
23:   cmp eax, ecx
      jge 30

26:   mov [esi], eax
28:   mov [esi + 4], ecx

30:   inc edi
31:   cmp edi, edx
      jlt 15
```

Generated native code

Figure 3.4: Code translation in IBM-JIT

Performance

The IBM-JIT compiler was tested on the SPECjvm98 benchmark. The comparison between the IBM-JIT compiler and the Sun's reference implementation of JDK 1.1.7, on a machine with a pentium II 350 MHZ and a 512 MB memory, proves the high performance of the IBM-JIT compiler. The execution time acceleration is more than 50 % for some tests (on Jack and Javac compilers). Figure 3.5 outlines these results. Higher bars mean higher performance. The results are relative to the IBM-JIT compiler.

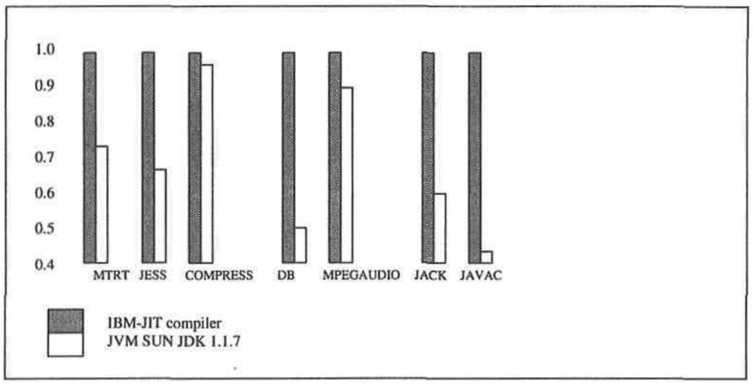


Figure 3.5: IBM-JIT performance

The IBM-JIT compiler performs good optimizations. These optimizations are based on some data flow analyses. These analyses require a lot of memory. However, in an embedded context, lightweight optimizations should be designed to fit memory constraints.

3.4.3 JUDO JIT Compiler

Ciernak et al. [25] designed a JIT compiler. Their work deals mainly with Java Under Dynamic optimizations (JUDO). The compiler architecture is composed of three components: a fast code generator, an optimizing compiler and a profiler. Figure 3.6 shows these components.

The fast code generator produces an unoptimized native code. It operates in two passes: the first is for building basic blocks, the second one is for performing some lightweight optimizations and inserting profiling functions.

The profiling data is a collection of method entry points to detect call-intensive

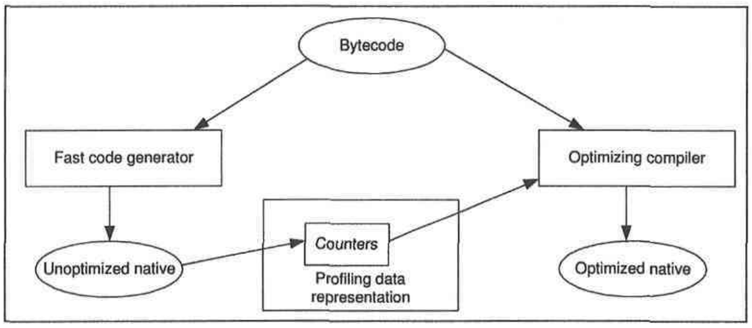


Figure 3.6: JUDO components

methods and back edge information to detect loop-intensive methods; the latter are considered very JIT-sensitive. These data are useful for deciding if the method is hotspot. In this case, it could be recompiled by the optimizing compiler.

The optimizing compiler is based on some global optimizations. Among these optimizations, copy propagation and dead code elimination are performed. Then, a global register allocation is performed. In this step, the native code contains just the machine registers that are available. The code scheduling phase reorders the native instructions to get a better performance. The structure of the optimizing compiler is depicted in Figure 3.7.

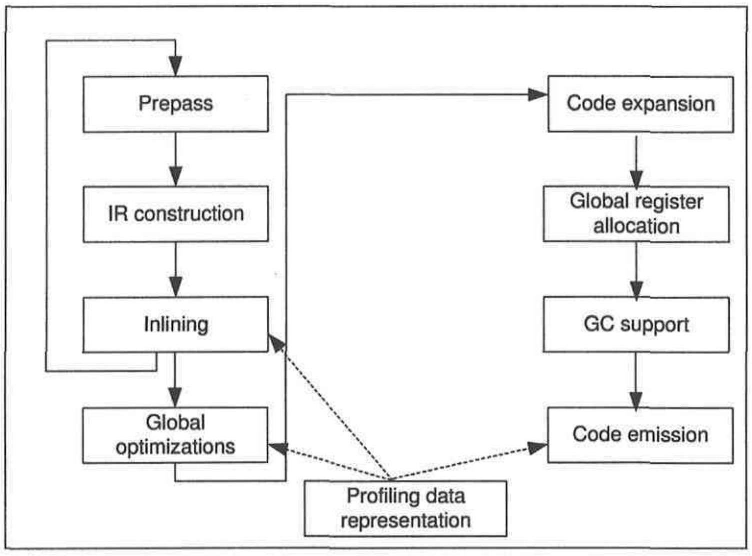


Figure 3.7: JUDO optimizing compiler structure

Static Optimizations

Some static optimizations are implemented in this work. For instance, array bound checks can be eliminated if the compiler is sure that the index is inside the array bounds. This is performed for array access in loops. A simple loop that does not contain an array bounds check is inserted in the code when the compiler proves that the index is inside the array bounds range. This loop duplication could trigger a code explosion.

Inlining is one of the performed optimizations. A method is inlined if it is called frequently and its code size is not greater than 25 bytes. For further calls, an inlined method is tested to know if its code is the actual one to execute. A simple equality test between the virtual table of the receiver object and that of the inlined method class provides such a decision.

Dynamic Optimizations

Some dynamic optimizations are implemented in this work. A dynamic inline patching (an inlining that takes into consideration the dynamic class loading problem), a lazy garbage collection and a lazy exception mechanism are among these optimizations. For the lazy exceptions mechanism, the exception object is not always created. This allows to save in some cases the memory space. Removing the exception object creation is possible only when this creation has no side effects on the variables of the program.

Dynamic Recompilation

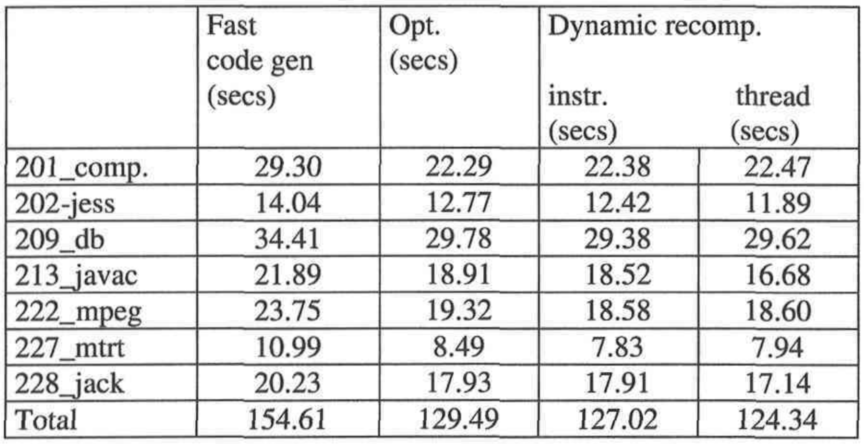
A *dynamic recompilation* mechanism is used to recompile methods that are hotspot. Two mechanisms are introduced in order to perform such a recompilation: a profiling mechanism and a thread-based mechanism.

In the first mechanism, the code is instrumented and counters are associated to methods. The recompilation is triggered automatically when a counter becomes zero. Choosing the value of a counter is an issue since a low counter triggers the recompilation of less frequently called methods. A high counter value leads to the recompilation of frequently called methods and could neglect less frequently called ones.

In the thread-based mechanism, a thread scans periodically the profiling information to know when the recompilation should happen. The drawback of this strategy is that all the methods are scanned and that a hotspot method could be detected late.

Performance

Figure 3.8 shows the JUDO performance. The first column gives the execution times obtained when running each of row tests by the fast code generator. The second gives the execution times when the optimizing compiler is included and the last two columns deal with the execution times when the dynamic recompilation technique is applied.



	Fast code gen (secs)	Opt. (secs)	Dynamic recomp.	
			instr. (secs)	thread (secs)
201_comp.	29.30	22.29	22.38	22.47
202-jess	14.04	12.77	12.42	11.89
209_db	34.41	29.78	29.38	29.62
213_javac	21.89	18.91	18.52	16.68
222_mpeg	23.75	19.32	18.58	18.60
227_mtrt	10.99	8.49	7.83	7.94
228_jack	20.23	17.93	17.91	17.14
Total	154.61	129.49	127.02	124.34

Figure 3.8: JUDO performance

The results show that the dynamic recompilation technique gives good results except for the first test because the profiling information did not help much to improve execution time. JUDO has the same disadvantages as the HotSpot and the IBM-JIT, which concern time and memory requirements.

3.4.4 LaTTe

LaTTe [139] is a JIT compiler that translates a method to native code the first time it is invoked. The architecture of LaTTe is depicted in Figure 3.9.

LaTTe performs bytecodes translation in five steps:

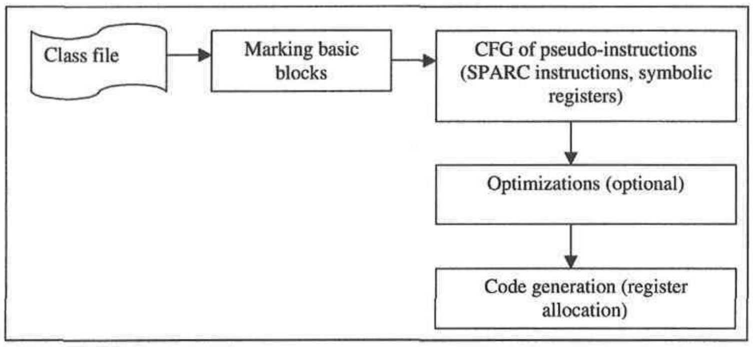

Class file
Marking basic blocks
CFG of pseudo-instructions (SPARC instructions, symbolic registers)
Optimizations (optional)
Code generation (register allocation)

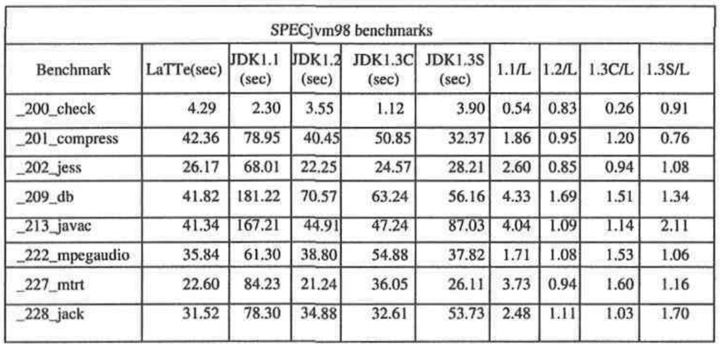
Figure 3.9: LaTTe architecture

1. Branch instructions and basic blocks are computed.
2. A translation of bytecodes into pseudo code (code that uses symbolic registers) is performed: the pseudo code contains symbolic registers. For instance, the bytecode `iaload n` is translated into the native instruction `mov il{n}, is{TOP + 1}`, which consists of copying the $\mathtt{n}^{\mathtt{th}}$ variable to the stack. `il` refers to the local variables table and `is` refers to the stack. These are symbolic registers.
3. Some optimizations are performed. This is an optional step.
4. Fast register allocation: the register allocation technique used by LaTTe is efficient. In fact, tree regions are extracted from the CFG of pseudo-instructions. A tree region is a single entry and multiple exit sub graph. A fast backward and forward sweep algorithm is performed on these regions to allocate machine registers to symbolic registers. A CFG of real SPARC instructions is generated (instructions that use real registers).
5. Native code emission is performed.

Performance

LaTTe performance is presented in Figure 3.10. Columns give the execution times of several tests in seconds. These tests were performed on a Sun UltraSPARC-IIi 333MHz processor with 128 MB of memory.

The results show that LaTTe improves the execution speed of the SpecJVM98 programs by a factor of 4 comparatively with the JDK 1.1 and up to 2 with the JDK 1.2 and the JDK 1.3.



SPECjvm98 benchmarks									
Benchmark	LaTTe(sec)	JDK1.1 (sec)	JDK1.2 (sec)	JDK1.3C (sec)	JDK1.3S (sec)	1.1/L	1.2/L	1.3C/L	1.3S/L
_200_check	4.29	2.30	3.55	1.12	3.90	0.54	0.83	0.26	0.91
_201_compress	42.36	78.95	40.45	50.85	32.37	1.86	0.95	1.20	0.76
_202_jess	26.17	68.01	22.25	24.57	28.21	2.60	0.85	0.94	1.08
_209_db	41.82	181.22	70.57	63.24	56.16	4.33	1.69	1.51	1.34
_213_javac	41.34	167.21	44.91	47.24	87.03	4.04	1.09	1.14	2.11
_222_mpegaudio	35.84	61.30	38.80	54.88	37.82	1.71	1.08	1.53	1.06
_227_mtrt	22.60	84.23	21.24	36.05	26.11	3.73	0.94	1.60	1.16
_228_jack	31.52	78.30	34.88	32.61	53.73	2.48	1.11	1.03	1.70

Figure 3.10: LaTTe performance

LaTTe does not use a hotspot strategy, hence needing a high memory footprint. Moreover, the translation of all the methods is not a good strategy since the translation time of the less frequent code will decrease the overall performance.

3.4.5 Aware Just In Time (AJIT) Compilation System

Azevedo et al. [10] designed a static compiler, which computes some annotations that simplify the JIT compiler work. An IR is generated from Java bytecodes. This IR is a three-address based representation. Some traditional optimizations such as common subexpression elimination and copy propagation are performed on this IR. The generated annotations help to perform garbage collection, runtime code optimization and register allocation.

Among these annotations, we cite Virtual Register Annotations (VRA). An example of VRA for the `iaload` opcode is depicted in Figure 3.11. VRA are composed of four fields: `SRC-SRC-EXTRA-EXTRA-DEST`. The first field represents the virtual register for the array address, the second one represents the virtual register for the index, the third represents the value of the index, the fourth contains the value of the array address and the fifth represents the virtual register that will contain the array element that is read from the memory.

An unlimited number of virtual registers is used to perform a virtual register allocation. VRA are used by the JIT compiler to produce native code in an efficient way. The virtual register allocation process assigns the minimum number of virtual registers to the most important variables (e.g. the variables that are accessed frequently). This technique allows to allocate dynamically the minimum number machine registers to these variables. A run-time register allocation algorithm maps virtual registers to

Bytecode	**Java IR**
iaload	v_0 holds array address v_1 holds index 1 : ishl v_1, "ishift", v_2 2 : iadd v_2, "arraySizeOffset", v_2 3 : aadd v_0, v_2, v_3 4 : ild (v_3), v_4

Annotated Bytecode					
Opcode	SRC	SRC	EXTRA	EXTRA	DEST
iaload	v_0	v_1	v_2	v_3	v_4

Figure 3.11: VRA annotation for `iaload` opcode

machine registers. If machine registers are insufficient, virtual registers are stored in stack temporaries.

The main disadvantage of this technique is that the class file size could be increased dramatically. Moreover, the time to handle annotations could be high and it is added to the code compilation time. Besides, the source code of applications is not always available since code can be loaded dynamically.

Performance

The results are presented in Figure 3.12. Columns represent the execution times (in seconds). The results show that the execution of the native code produced by the AJIT compiler for a Java method is 4.83 faster than its interpretation under the JDK 1.1.1 interpreter.

Benchmarks	Sun Interpreter (secs)	AJIT (secs)
Neighbour 256x256 array Iterations = 1500	553.03	115.31
EM3D 1250 tree nodes Iterations = 200	359.84	74.51
Bitonic Sort 1024 tree nodes Iterations = 512	167.05	120.96
Huffman 30000 array nodes Iterations = 288	4690.00	1487.00

Figure 3.12: AJIT performance

3.4.6 Jalapeño Dynamic Optimizing Compiler

Burk et al [4] designed a Java JIT compiler called the *Jalapeño dynamic optimizing compiler*. Figure 3.13 shows the Jalapeño architecture. This architecture is similar to that of JUDO.

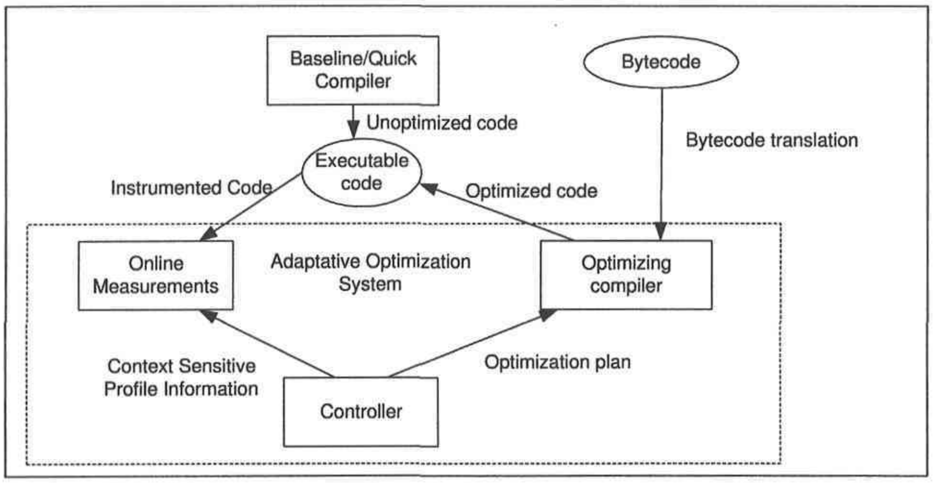


Figure 3.13: Jalapeño architecture

The baseline compiler generates an unoptimized native code. The generated code provides slightly better performance than the interpreted code. The baseline compiler does not perform register allocation. The Jalapeño compiler uses three IRs:

- A High-level Intermediate Representation (HIR): in this representation, an instruction is composed of an operator and a number of operands. An operator is a reserved word like an opcode. Jalapeño adds explicit operators that are related to dynamic verifications such as `NULL_CHECK` that verify if a pointer is null or `BOUNDS_CHECK` that verifies if an array index is in a specific range. Table 3.2 shows a Java program, its Jalapeño IR and the corresponding optimized representation. This example shows how the optimized representation removes unnecessary temporary variables used in the IR representation.

- A Low-level Intermediate Representation (LIR): HIR is translated into LIR, which is a low-level representation that is specific to the Jalapeño. Some specific opcodes such as `invokevirtual` or `getfield` are translated into some operations that use explicitly object and method table layout.

- A Machine-specific Intermediate Representation (MIR): a dependance graph is built from LIR. LIR contains temporaries and local variables. A register allo-

Java bytecode	Intermediate Representation	Optimized representation
`iload_x`	`INT_ADD tint, xint, 5`	`INT_ADD yint, xint, 5`
`iconst_5`	`INT_MOVE yint, tint`	
`iadd`		
`istore_y`		

Table 3.2: A Jalapeño optimized HIR

cation step is performed on this representation. The priority in allocating machine registers is given to temporaries that are live just in the current basic block. Jalapeño uses a linear scan of the variables liveness range [104] to perform register allocation. This algorithm is proved to be faster than a coloring graph algorithm.

Jalapeño defines three levels of optimizations:

- Level 0: includes optimizations that are performed when Java bytecodes are translated into an intermediate representation. Among these optimizations, we cite constant and copy propagation.
- Level 1: includes more local optimizations such as common subexpression elimination.
- Level 2: adds more optimizations to the level 1. These optimizations are performed on programs that are in a Static Single Assignment form (SSA). In the SSA representation, each program variable appears on the left hand side of only one assignment statement. Many analyses are easier to be performed on the SSA representation.

Besides, Jalapeño has an architecture for dynamic selective compilation. This architecture has three components:

- Runtime measurements subsystem: can support many measurement techniques such as method call counters, basic blocks and back edges profiling.
- Controller: takes into consideration the measurement information in order to make decisions about the methods that should be recompiled.
- Recompilation subsystem: it is composed of compilation threads that invoke the optimizing compiler. This compiler performs a recompilation of the methods that are chosen by the controller.

The recompilation subsystem is composed of:

- Optimization Plan: contains the optimizations to perform.
- Profiling data: contains data that help to perform optimizations.
- Instrumentation plan: contains instrumentation instructions that will be inserted in the native code.

For Jalapeño, only the methods that are identified to be hotspot are recompiled. A counter is associated with each method. At a call site, both counters of the called and the caller method are incremented. A method counter is also incremented in a back edge program point. A thread scans method counters every 10 milliseconds. When a method counter exceeds a threshold, the method is considered to be frequently called. The recompilation decision depends on some parameters. Let:

- i be the current optimization level.
- j be the level to which a method compilation decision will be taken. We suppose that $j > i$. Let m be this method.
- T_i be the future time spent by the program if the method m is not compiled.
- C_j be the cost in time of recompiling m at the level j.
- T_j be the future time spent if m is recompiled at level j.

m is recompiled at level j if $C_j + T_j < T_i$. The future time estimation of an execution is an issue. Jalapeño supposes that if a program spends x seconds in execution, it will spend x seconds in the future. Let P_m be the estimated percentage of future time in a method m and T_f the future expected running time of the program. The estimated future execution time of m is $T_i = T_f * P_m$. Let S_k be the estimated speedup at level k compared to level 0. The future time spent, if a recompilation happens at level j, is $T_j = T_i \times S_i \ / \ S_j$.

Register Allocation

Jalapeño uses a linear scan algorithm to perform register allocation. This algorithm has been designed by Poletto and Sarkar [104]. It is more efficient than a graph coloring algorithm. This algorithm operates as follows:

- Order the instructions.
- Compute the set of live intervals for the code variables.
- Allocate a virtual register to each interval:
  - If a register is available then allocation is possible.
  - Otherwise, a previously allocated register is chosen. Then, the spill code is added. This choice depends on the live intervals of the registers.
- Rewrite the code according to the allocation:
  - Machine registers will replace virtual ones.
  - Spill code is generated.

Jalapeño Performance

The Jalapeño baseline compiler is two times faster than the IBM JVM 1.1.8. The optimizing compiler is as fast as the IBM JIT for this JVM. The high memory footprint is the major drawback of the Jalapeño compilation strategy. Generating native code for all the methods (the baseline compiler strategy) is a bad strategy because there is a waste of time and memory in compiling less frequent methods. However, a one-pass strategy is interesting for our work since it is an efficient way to generate native code.

3.5 Dynamic Compilation Challenges in an Embedded Context

The studied dynamic compilers for desktop and server systems require an important space memory to store intermediate representations and native code. The required memory can go beyond some Megs, which is unaffordable in embedded platforms. Another common feature of these compilers, is the need for many passes to perform heavy analyses. This is time-consuming and harmful for limited-power devices.

We are convinced that there are two key elements for the success of a dynamic compilation in an embedded context:

- The compilation should be efficient. Performing many passes induces a high power cost and does not guarantee that, in the future, the spent time will be compensated.
- The compilation should be aware of memory constraints. Compact data structures should be designed to minimize footprint. Moreover, the compiler size should be minimized since it is accounted in the global overhead.

For the design and implementation of our embedded dynamic compiler (that we called E-Bunny), we set the following requirements:

- The compilation should be fast and lightweight. Heavy optimizations are avoided and the compiler should use the available dynamic information to get a native code of good quality.
- Because of memory constraints in embedded systems, it is not possible to compile all bytecodes, so we choose to compile a selected set of methods. Moreover, the unit of compilation is a method. The rationale behind this is to avoid heavy and more frequent switching mechanisms when the unit is a set of instructions. Our design will target Connected Limited Devices (with memory space less than 512 KB) so memory footprint should not exceed 150 KB.
- A cache is used to store the generated machine code. We fixed the maximum size that the cache can reach to be 64 KB. This is a reasonable size for embedded Java platforms with 512 KB of available memory.

Our strategy rests on establishing a tradeoff between time compilation and the quality of the generated code. Detailed technical description of this strategy is presented in the next chapter.

3.6 Conclusion

We have investigated many dynamic compilers for desktop and server class systems. The high compilation overhead is the main drawback of these compilers. This is unacceptable for embedded low-power devices. Moreover, the memory footprint of these compilers is high and is surely inappropriate for embedded platforms with less than 512

KB of available memory (like J2ME/CLDC). As mentioned above, embedded VM acceleration projects were leaded mainly by companies so the state of the art in such research field is poor. Hence, our first challenge is to accelerate Java in an embedded context by following two lines: dynamic compilation techniques and general optimizations techniques. These techniques are carefully designed in order to take into consideration embedded system constraints.

Chapter 4

E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines

4.1 Introduction

In this chapter we present an extremely lightweight dynamic selective compiler for embedded Java virtual machines called E-Bunny. This compiler is built on top of KVM. The contributions of this work are threefold:

- Our research is the first academic initiative that targets CLDC-based embedded Java virtual machines optimization by dynamic compilation. The remaining projects such as CLDC HotSpot [84] and Jbed [113] are commercial.
- Our solution, besides the compilation of all kinds of bytecodes, covers the different issues of the integration of a dynamic compiler into a virtual machine such as multi-threading support, exception handling, garbage collection, switching mechanism between the compiler and the interpreter modes, etc.
- Our solution is efficient. It allows to improve the KVM performance by a rate of 400% while the memory footprint overhead does not exceed 138 KB.

4.2 Architecture

E-Bunny is a dynamic selective compiler for embedded Java virtual machines that uses as its foundation the KVM. In this section, we present the key features that make E-Bunny an appropriate Java acceleration technology for embedded systems. The major features of E-Bunny are:

- **Reduced Memory Footprint**: the footprint resulting from the integration of the E-Bunny dynamic compiler does not exceed 150 KB. The key idea of reducing the code size of E-Bunny is carried out by merging the compilation processing of some bytecodes. This is possible because several bytecodes have common processing (e.g. invokespecial, invokevirtual). This strategy is applied mainly for some quickened forms of bytecodes [72](e.g. getstatic, getstatic_fast, getstaticp_fast, getstatic2_fast).
- **Selective Compilation**: since dynamic selective compilation is the most adequate compilation-based acceleration technique for embedded systems, it was adopted in E-Bunny. Only a subset of methods is compiled. The methods are selected according to their invocation frequencies. The unit of compilation is exclusively a method. The rationale behind this decision is to minimize the technical complexity of the switching between interpreted and compiled codes.
- **Efficient Stack-Based Code Generation**: for the compilation strategy, a trade-off has to be made between the compilation cost and the generated code quality. Although a register-based code is more efficient, we do not generate such code because it requires more passes over the bytecode and increases considerably the compilation time. In E-Bunny, we generate a stack-based code because it requires only one-pass over method bytecodes. Thus, a one-pass code generation strategy is adopted, without using neither intermediate representations nor heavyweight optimizations. Only optimizations that might be applied in one-pass are allowed.
- **Multi-Threading Support**: another challenge introduced by dynamic compilation is the multi-threading support. Conventionally, each thread has its own execution stack. Since our approach uses two kinds of stacks, each thread is assigned two stacks upon its creation: a Java stack to interpret methods and a native stack to run compiled ones. For the Java stack, we keep the KVM management strategy in which the KVM allocates the Java stack in the heap and manipulates it at a software level. For the native stack, the adopted approach is

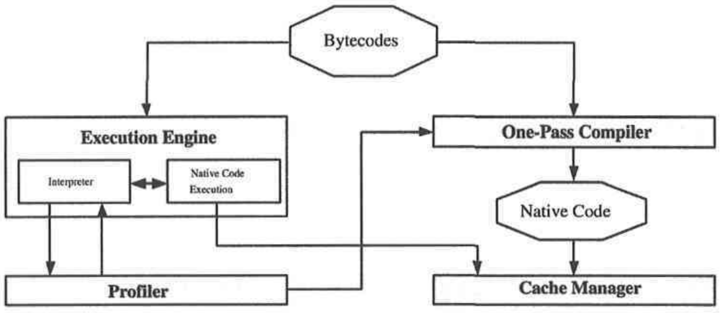


Figure 4.1: E-Bunny architecture

to organize the native stack as a pool of segments and allocate a segment to each thread. Thus, the native stack will be shared by all living threads.

- **LRU Algorithm for Cache Management**: a limited memory space is allocated to the compiled code. When this space is full, a cache strategy based on a Least Recently Used (LRU) algorithm [74] is adopted to free the necessary space.

E-Bunny architecture is depicted in Figure 4.1. It includes four major components: the execution engine, the profiler, the one-pass compiler and the cache manager. Initially, all invoked Java methods are interpreted. During interpretation, a counter-based profiler gathers profiling information. As the code is interpreted, the profiler identifies hotspot methods. Once a method is recognized as hotspot, its bytecodes are translated into native code by the compiler. The produced native code is stored in the dynamic compiler cache. On future references to the same method, the cached compiled method is executed instead of interpreting it. In the sequel, we highlight the main components of the proposed embedded dynamic compiler architecture.

4.2.1 Interpreter

By interpreter, we mean the KVM's interpreter, which is basically a loop that fetches, decodes and interprets bytecodes of a given method. E-Bunny adopts a mixed-mode[1] execution approach. Therefore, the interpreter cooperates with the native code execution component to switch between two modes: interpreted and native.

[1]Execution overlaps between interpretation mode and native code execution mode

4.2.2 Native Code Execution Component

The native code execution component is responsible for invoking compiled methods. Basically, it looks for the corresponding native code in the cache and then executes it. In addition, this component is responsible of transferring method arguments, if any, from one stack to the other. More details about this mechanism are given in Section 4.3.2. When the native code of a given method is executed, the cache must be set up-to-date according to the cache algorithm. The native code execution component triggers the cache update.

4.2.3 Profiler

A simple counter-based profiler is used in E-Bunny. A per-method counter is incremented each time the method is invoked. A method is considered as hotspot when its counter reaches a threshold. This threshold is known after running tests on the accelerated KVM. These tests allow to know the threshold that gives the best speedup. For instance, for the CaffeineMark benchmark, the threshold is 200. Consequently, a compilation request is triggered. Our profiling strategy assumes that every method called by a compiled method must be compiled. So, a compiled method cannot invoke an interpreted method. The motivation behind this strategy is to reduce the complexity of the switching mechanism between interpreted and native modes.

4.2.4 One-Pass Compiler

The one-pass compiler is the core component of E-Bunny. It is triggered by the profiler. Basically, it compiles Java methods to native code. The E-Bunny compiler is extremely lightweight. It goes through the bytecode in one-pass and generates a stack-based code. The detailed compilation strategy is described in Section 4.3.1.

4.2.5 Cache Manager

Once a method is recognized as hotspot and compiled, the corresponding native code needs to be stored in the heap. In E-Bunny, a memory space, called the cache, is pre-allocated in the heap to store the generated code. The cache is pre-allocated in the permanent space of the heap to avoid any conflicts with the garbage collection mechanism. Conventionally, the garbage collector does not scan the permanent space. When the cache becomes full, the cache manager must select elements to remove in order to free space for newly compiled methods. For this purpose, an LRU algorithm is used. This algorithm selects the methods that have not been invoked for the largest period of time and removes them from the cache. A queue is used to keep the chronological order of invoked methods. This queue is updated each time a compiled method is invoked. The LRU algorithm does not prevent methods from being recompiled several times, but our experiments show that using an LRU algorithm reduces efficiently the re-compilation rate.

4.3 Design

In this section we discuss the design issues of E-Bunny. First, we detail the compilation strategy. Second, we focus on a delicate aspect of dynamic selective compilation, which is the switching mechanism between interpreted and compiled modes. Then, we illustrate how E-Bunny supports multi-threading. Finally, we describe the interaction with the garbage collection mechanism.

4.3.1 Compilation Strategy

Our compilation strategy spans over a lightweight one-pass compilation technique. This strategy avoids complex computations performed by common compilers and generates a code of reasonable quality. Indeed, the generated code is stack-based as Java bytecode but uses many information computed at the compilation step (field offsets, constant pool entry address etc.). These information are grafted in the generated code in order to avoid unnecessary further re-computation.

Compiling a method goes through three steps. First, generating context saving instructions (the prologue). Second, translating bytecodes into native code instructions. Third, generating context restoration instructions (the epilogue). The second step, which is the core step of our compilation strategy, consists of translating each bytecode into a sequence of native instructions. We distinguish two categories of bytecodes. The first kind includes bytecodes that are completely translated into native instructions in a straight manner. They are called: simple bytecodes. The second kind of bytecodes are more complex to translate and are called complex bytecodes. E-Bunny targets Intel

```
//saving old value of EBP on the native stack
push EBP
//setting the new value of EBP
mov EBP, ESP
//CurrentThread->LastEBP = EBP
mov EAX,&CurrentThread->LastEBP
mov [EAX], EBP
```

Figure 4.2: Context saving

IA-32 architecture [29]. This version was successfully ported to ARM architecture by another member of the team [11]. IA-32 is a 32 bit CISC machine, which provides eight general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP). EBP and ESP are dedicated to stack management. The remaining registers fulfill different other tasks. Hence, compiling a method consists of generating assembly instructions that reproduce the bytecode behavior on an IA-32 platform.

Context Saving and Context Restoration

Context saving and context restoration instructions are used to re-establish a calling method context after the execution exits from a called method. Context saving instructions figure on top of the method generated code. They carry out three basic operations. First, save old EBP register value (of the calling method) on the native stack. Second, assign a new value to EBP register (of the callee method). Third, assign the new value of EBP to *LastEBP* field of the current thread structure (*CurrentThread->LastEBP* =EBP). Actually, *LastEBP* is a new field added to the thread data structure in E-Bunny. It gives for each thread the EBP value of the last called compiled method. This information is used for two purposes:

1. Garbage collection: the object references in the native stack should be checked when a garbage collection happens.
2. Exception handling: exceptions could be propagated from a compiled method to another one.

Context saving instructions in E-Bunny are illustrated in Figure 4.2.

Context restoration instructions figure on the end of method generated code. Since all Java methods finish by a return bytecode (ireturn, lreturn, areturn or return), con-

```
//restore the old value of EBP register
mov ESP, EBP
pop EBP
//CurrentThread->LastEBP = EBP
mov EAX,&CurrentThread->LastEBP
mov [EAX], EBP
```

Figure 4.3: Context restoration

```
iload (0x15)
  Description: Load integer from local variable
  Format: iload <index>
  Corresponding Assembly Instructions :
    push [EBP + 4 * ( <FrameSize> - <index> + 3)]
```

Figure 4.4: iload bytecode translation

text restoration instructions are generated when translating return bytecodes. Context restoration instructions carry out two operations. First, restore the old value of EBP register (saved on the native stack). Second, assign this value to *LastEBP* field of *CurrentThread* data structure. Consequently, when returning to the calling method, the EBP register and *CurrentThread->LastEBP* are set to the appropriate values. Context restoration instructions in E-Bunny are illustrated in Figure 4.3.

Simple Bytecodes Translation

The simple bytecode category includes loads (e.g. iload, iaload, ldc), stores (e.g. astore, lastore), stack manipulation (e.g. pop, dup), arithmetic, logic and shift (e.g. iadd, land, ishr, l2i) and branching (e.g. ifne, if_icmpeq, goto) bytecodes. These bytecodes are directly translated into a stack-based native code, which reproduces the interpreter behavior on the native stack. As an example, we give the translation details of two bytecodes: iload and ifeq. We have selected a stack manipulation bytecode (iload bytecode) to show how it behaves on the native stack and a branching bytecode (ifeq) to illustrate how E-Bunny translates forward branching instructions in one-pass.

Stack Manipulation Bytecodes Translation

The bytecode iload loads an integer (32 bit value) from a local variable to the top of the stack (stack operand). It is directly mapped to a push instruction as shown in Figure 4.4. Note that the symbols "[ ]" denote the content of the specified register and the FrameSize variable is an integer that gives the number of local variables of the

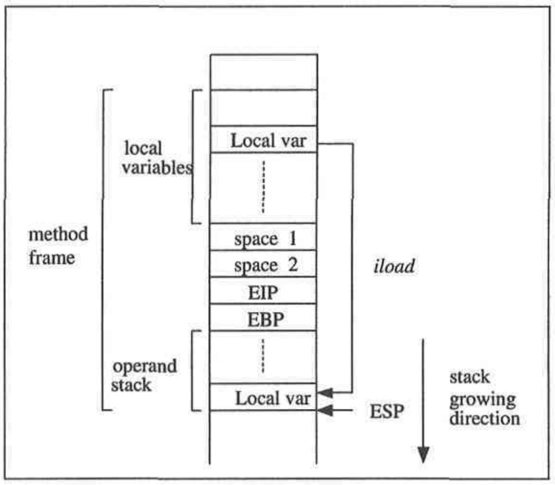


Figure 4.5: iload on the native stack

current method. The value 3 in the offset of the push instruction denotes the space, in bytes, left for EIP register and for two empty spaces for the returned value (space 1 and space 2) as illustrated in Figure 4.5.

Control Flow Bytecodes Translation

An important issue of the one-pass translation strategy is how to deal with control flow bytecodes (or branching bytecodes). In the sequel, we highlight this issue and how it is processed in E-Bunny.

Control flow bytecodes are translated using a bytecode map table called *BCNativeMap*. It is an integer table that maps each bytecode index to its corresponding native instruction index. For instance, *BCNativeMap*[5] = 31 means that the native instructions corresponding to the fifth bytecode start at index 31 in the *NativeCode* table. A *BCNativeMap* table is associated with each compiled method. It is initialized to zero and filled progressively when the translation of the method is being performed. At a given moment of the translation, the *BCNativeMap* table is filled for the scanned bytecodes and empty (zero values) for the rest.

When we deal with control flow bytecodes, there are two situations that should be handled: forward branches and backward branches. The translation of backward branches is simple. In fact, the *BCNativeMap* table is used to get the target native instruction index and then to generate the corresponding jump native instruction. However, the translation of forward branches is more complex. Indeed, the corresponding *BCNativeMap* entry of a forward branch is not resolved yet (contains zero) since the target bytecode is not compiled yet. The complete translation of the forward branching bytecode is postponed until the target bytecode is reached. Actually, a jump native instruction opcode (e.g. jl, jne, jmp) is generated with an unresolved operand. Each ifeq (0x99)

Description: Branch if equal to zero

Format: ifeq <offset>

PseudoCode:

```
if (BCNativeMap[currentBytecode + offset] == 0 )
              //forward branching//
    generate a jump instruction without operand
    leave space for the operand
else          //backward branching//
    generate jump instruction
    compute offset operand
    generate operand
endif
translate next bytecodes
when reaching the target bytecode:
    compute the current offset
    fill the space left before by the computed offset
```

Corresponding Assembly Instructions:

```
    pop EAX
    cmp EAX, 0
    jz <computed native code offset>
```

Figure 4.6: ifeq bytecode translation

next bytecode is checked whether it is a target of an unresolved branching. In such case, the corresponding offset is computed and then used to update the unresolved operand of the jump native instruction. To handle multiple unresolved forward branching, a linked list is used. In Figure 4.6, we give the translation algorithm of the bytecode ifeq and the corresponding native instructions.

Complex Bytecodes Translation

The KVM interpretation of some bytecodes involves the use of some virtual machine runtime services such as method lookup or field reference resolution. For example, the bytecode getfield uses a field reference resolution subroutine to compute the reference of the appropriate field. In E-Bunny, bytecodes that require virtual machine services are called complex bytecodes. Translating this kind of bytecodes is more complex. A possible approach consists of generating the corresponding native code, instruction by instruction, including virtual machine services. This yields a complex and a very bulky code. The approach adopted in E-Bunny is to define for each bytecode a specific function that calls necessary virtual machine services. These functions are called from the native code. Hence, the generated native code is compact and less complex. Complex bytecodes category includes field access, objects creation, array manipulation, method invocations, return, monitor access, casting and exception bytecodes. In the sequel, we illustrate the translation mechanism of field access bytecodes and the exception bytecode (athrow).

Field Access Bytecodes Translation

Field access bytecodes are putfield, getfield, putstatic, getstatic and their quickened versions. These bytecodes make access to object fields using symbolic references, to get or set their values. Such symbolic references have to be resolved so that the access could be possible. In E-Bunny, for each bytecode a specific function is defined. For getfield and getstatic, this function calls the field reference resolution subroutine and returns the value of the field in EAX register (EAX and ECX for long values). This value is then pushed into the native stack. For putfield and putstatic, this function calls the field reference resolution subroutine, sets this field and returns the field size in EAX register. This value is then used to update the native stack. The translation of putfield is given in Figure 4.7. Note that cp_index denotes a constant pool index. The "switch" operator is used to clarify the illustration. Actually, it is implemented with jump instructions.

Method Invocation Bytecodes Translation

Method invocation bytecodes require a special care because they involve more complex processing than other bytecodes. Furthermore they are performance-critical (on average 11% of total bytecodes are method invocation bytecodes [109]). The processing, putfield (0xb5)

Description: Set field value in object

Format: putfield <cp_index>

Function used: putField()

Pre-conditions:

the following items must be on the top of native stack:

- ip (bytecode ip)
- field value
- objectref.

Post-conditions:

set EAX to the filled size:

1 when the field needs one memory word

2 when the field type is two memory words

Corresponding Assembly Instructions:

```
//push the ip into the Intel stack
push ip
mov ECX, &putField
call ECX
    //popping the putfield arguments
    //switch// ECX
    1 : add ESP, 12
    2 : add ESP, 16
```

Figure 4.7: putfield bytecode translation

that method invocation bytecodes should carry out, includes:

- Performing a method reference resolution.
- Checking the validity of the receiver object.
- Performing a dynamic method lookup.
- Creating a new frame for the invoked method.

Moreover, these bytecodes deal with the switching mechanism between interpreted and compiled modes (more details about the switching mechanism are given further).

In E-Bunny, to translate method invocation bytecodes, specific functions are defined (callVirtual for invokevirtual and invokespecial, callStatic for invokestatic and callInterface for invokeinterface). These functions behave differently according to the kind of the called method (compiled or native). The conventions adopted in the design of these functions are the following: the defined functions resolve the invoked method reference, check the validity of the receiving object, make the method lookup and then:

- If the invoked method is native:
  1. Transfer the arguments from the native stack to the Java stack (because the called native method needs to get its arguments from the Java sack).
  2. Invoke the native method and finally transfer the returned value to the Java stack.

  In addition, these functions, set EAX register to 1 (to distinguish the native method call from the compiled method invocation) and EDX register to an integer value that is required to update the native stack (see Figure 4.8).

- If the invoked method is compiled:
  1. Get the native code from the cache.
  2. Update the latter and exit.

  Note here that the defined functions do not call the invoked method. Instead, the native code address of the called method and the local variables number are returned respectively in EDX register and in the top of the native stack. These values are used to prepare the invocation. After that, the method is invoked. When the execution exits from this method, the method context information is updated using the size of the value computed by the translated return bytecode.

invokevirtual (0Xb6)

Description: Invoke an instance method

Format: invokevirtuall <cp_Index>

Function used: callVirtual()

Pre-conditions:

the following items must be on the top of the Intel stack:

- ip (bytecode's ip)
- one empty word

Post-conditions:

```
if (called method is a C (native) function)
    - EAX = 1
    - The returned value is on the native stack
    - EDX = integer value to update the native stack.
else    //called method is a compiled java method
    - EAX = 0
    - [ESP] = local variables number
    - EDX = native code address.
endif
```

Corresponding Assembly instructions:

```
sub ESP,4        //leaving the first space
push ip          //pushing ip (and the second space)
mov EAX, &callVirtual
call EAX
  //Switch// EAX
  1 : add ESP, EDX      //updating the native stack
  0 : mov EAX, [ESP]    //getting space for locals
      sub ESP, EAX      //leaving space for locals
      call EDX          //launching the native code
      add ESP, EAX      //updating the native stack
      POP_FRAME_FROM_NATIVE
```

Figure 4.8: invokevirtual bytecode translation

athrow (0xbf)

Description: Throws an exception or error

Format: athrow

Function used: throwExc()

Pre-conditions:

the following item must be on top of the Intel stack:

- exception object reference

Post-conditions:

- jumps to the exception handler if any
- raise a virtual machine exception otherwise.

Corresponding Assembly Instructions:

```
mov EAX, &throwExc
call EAX
```

Figure 4.9: athrow bytecode translation

To illustrate this mechanism, the translation of the invokevirtual bytecode is given in Figure 4.8.

Notice that the method call preparation step and mainly the method lookup are not accelerated by the dynamic compilation. We propose, in the next chapter, a technique for accelerating this feature. Enhancing this mechanism allows an additional acceleration of method calls.

Exception Bytecodes Translation

Another important feature of the Java language is the exception handling mechanism. An exception is raised by the athrow bytecode. A major constraint that the dynamic compilation should respect is to preserve exception handling semantics. The dynamic compilation introduces new issues relevant to exception propagation. Indeed, a compiled method could propagate an exception to an interpreted method. E-Bunny handles the following situations:

- The method where the exception is thrown and the one where the exception is caught, are both interpreted: the original KVM exception handling mechanism is used.
- The method where the exception is thrown, is interpreted and the one where the exception is caught is a compiled method. This case is excluded since the adopted profiling strategy assumes that a compiled method cannot invoke an interpreted one.

- The method, where the exception is thrown, is a compiled method. There are two possible situations: the method catching the exception is either interpreted or compiled. In order to locate the method handling the exception, first, *BCNativeMap* is used to identify the bytecode corresponding to the native instruction throwing the exception. Then, an exception handler lookup is performed. If the method catching the exception (the method containing the handler) is a compiled one then its *BCNativeMap* is used to locate the native instruction corresponding to the bytecode handling the exception. A jump to this instruction is performed. Otherwise, we switch back to the interpreted mode at the level of this bytecode.

The generated code for the athrow bytecode calls a specific function called throwExc. This function uses the virtual machine exception handling functionality to locate the exception handler. Then, it resumes the execution at the appropriate location. The translation of athrow is illustrated in Figure 4.9.

4.3.2 Switching Mechanism

In E-Bunny, compiled methods are executed in the native stack while interpreted methods are executed in the Java stack located in the heap. Hence, execution of Java programs overlaps between the native and Java stack. The switching between the two modes implies context transferring between the two stacks. We distinguish two situations where the switching occurs between the two modes: interpreted to native and native to interpreted.

The switching from the interpreted to the native mode occurs when interpreting an invoke bytecode and the invoked method happens to be compiled. Coming from the interpretation mode, the called method arguments are on the top of the Java stack. The switching to the native mode requires their transfer to the native stack. This is carried out according to the algorithm in Figure 4.10. The algorithm is based on the method frame layout in the native stack, depicted in Figure 4.11.

The switching from the native mode to the interpreted mode occurs in two situations. First, when a compiled method calls an interpreted method. Second, when a compiled method exits and returns back to its interpreted caller method. The profiling strategy, we adopt, assumes that every method called by a compiled method should be compiled. The switching is then reduced only to the second situation (return case). Handling this switching consists of transferring the returned value, if any, from the native stack to the Java stack. The implementation of this action depends on the returned value type as illustrated in Figure 4.12.

Figure 4.11 shows the native stack in different phases of the switching. First, before a method call (a). Second, after the execution of algorithm 1 (b). Third, when the

```
ALGORITHM 1: interpreter to native switching
  Step1: transfer arguments
    for ($i <= NbrArguments$)
        push arg_i
  Step2: reserve space for local variables
  Step3: reserve two words for the returned value
    sub ESP, 8
  Step4: call the compiled method generated code
    call Method_Native_code
```

Figure 4.10: Interpreted to native mode switching algorithm

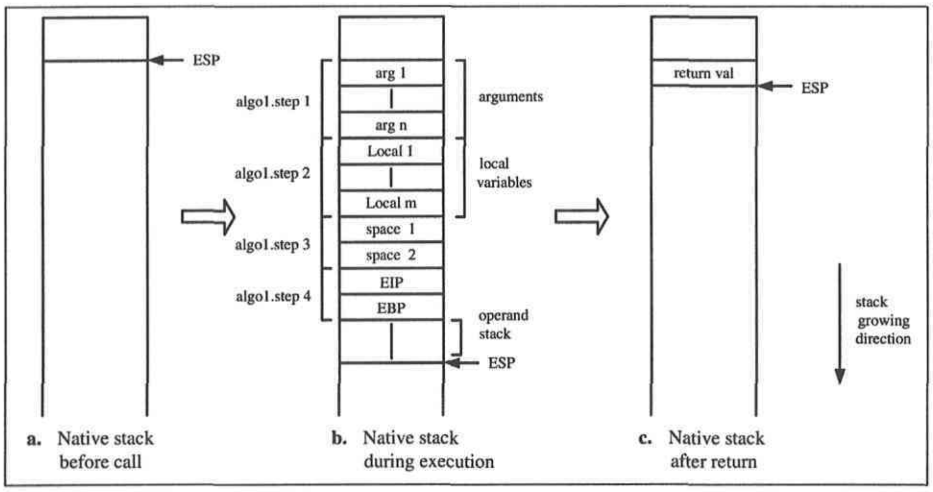


Figure 4.11: Switching to the native mode

```
ALGORITHM 2: native to interpreter return
 switch returned_value_type_size:
    0 //void
      nothing
    1 //integer, reference, etc.
      pop value
      pushStack(value)
    2 //long
      pop lowHalf
      pop highHalf
      pushStack(highHalf)
      pushStack(lowHalf)
```

Figure 4.12: Native to interpreted mode switching algorithm

translated called method exits (c). In (c), we assume, in this illustration, that the called method returns a one word size value (e.g. integer, short etc.).

It is worth mentioning that this switching hurts execution since the Java stack and the native stack are both manipulated. The next chapter proposes a new technique for enhancing the switching mechanism by unifying these stacks.

4.3.3 Threads Management

A Java virtual machine provides a framework to run properly different threads. Each thread has its own stack. In the interpreted mode, these stacks are created and managed at a software level. Basically, thread stacks are allocated in the heap. However, in E-Bunny, since two stacks are used, compiled methods have to be run on a native stack. Consequently, threads should use the native stack.

In the current implementation, the native stack is organized as a pool of segments. A segment is assigned to a thread when the latter is created. The segment pool management is based on a bit map. An entry of this map is a bit indicating whether the corresponding segment is used or free. Therefore, each thread executes its compiled methods in its own segment. A consequence of managing several threads with two stacks is that each thread has two forms of context information. The first is relevant to the Java stack (e.g. sp: stack pointer, fp: frame pointer, lp: locals pointer) and the second is relevant to the native stack (ESP, EBP and EIP registers). The data structure representing the thread in the virtual machine holds information representing both

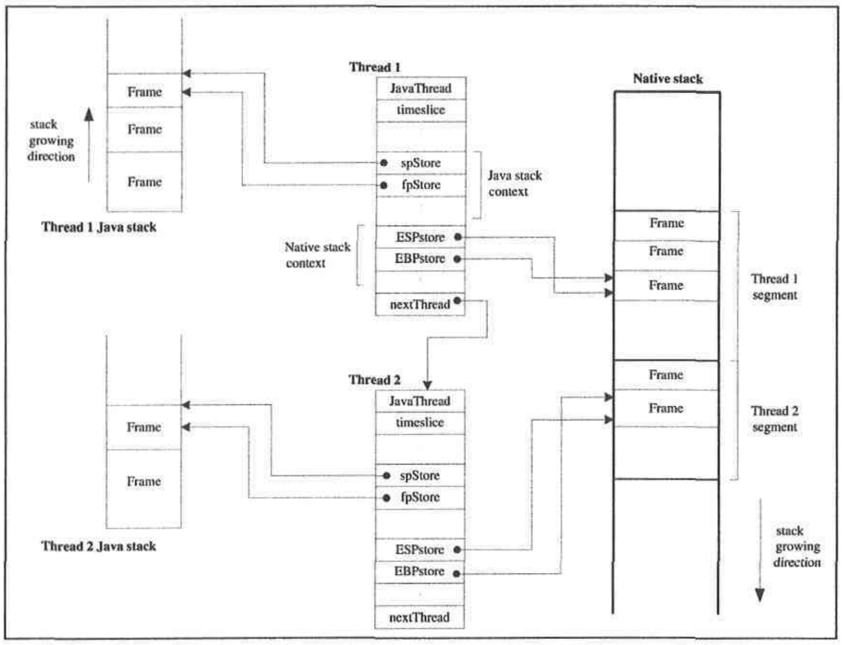


Figure 4.13: Multi-threading in E-Bunny

contexts (Figure 4.13).

Thread scheduling in the KVM is based on a round robin scheduling model. Each thread keeps control during a time-slice. This is decremented after each bytecode that may cause a control transfer (e.g. branching and invoke bytecodes). When the time-slice becomes zero, the virtual machine stops the current thread and resumes the next one in the running threads queue. Method compilation requires the support of thread scheduling. To achieve this purpose, additional code is generated for bytecodes that can cause thread switching. This code, mainly, decrements ESI register, dedicated to hold time-slice value and triggers thread switching when ESI reaches the null value. In addition, a special care to save both contexts relevant to Java and native stacks, is taken.

4.3.4 Garbage Collection Issues

KVM garbage collection is based on a mark-sweep with compaction algorithm. This algorithm goes through three steps: mark, sweep and compact. First, this algorithm scans all the heap in order to mark live objects. In particular, it scans method frames in Java stacks and marks referenced objects. Second, it sweeps the free chunks to constitute consistent blocks. Finally, an eventual heap compaction is carried out, leading to a move of the objects inside the heap. Object addresses are then changed and therefore,

```
Marking:
for each frame {
  if (compiled_method)
    Mark_Native_Stack_References(compiled_method)
  else
    Mark_Java_Stack_References(interpreted_method) }


Updating references:
for each frame {
  if (compiled_method)
    Update_Native_Stack_References(compiled_method)
  else
    Update_Java_Stack_References(interpreted_method) }
```

Figure 4.14: Garbage collection algorithm modifications

references to them, particularly in Java stacks, have to be updated.

With the selective approach of E-Bunny, compiled methods are executed on the native stack. Consequently, the native stack may contain some object references. Since the current garbage collection algorithm scans only the heap, the native stack will not be considered and then, object references on it will neither be marked nor updated. Therefore, the current garbage collection algorithm is inaccurate with a selective approach.

The garbage collection algorithm has to be extended to deal with the native stack. More precisely, translated method frames, which figure in the native stack, have to be scanned in order to mark and update object references. In E-Bunny, the garbage collection algorithm is enhanced to address this issue. Mainly, the garbage collection functionalities are modified to take into account the object references in the native stack. Indeed, for both marking and updating loops, we check if the frame corresponds to a compiled method or not. If the method is compiled we consider the native stack, otherwise we consider the Java stack. The modifications in marking and updating loops are illustrated in Figure 4.14.

KVM	E-Bunny	Footprint Overhead
144K	208 K	64 K

Table 4.1: Executable file footprint overhead

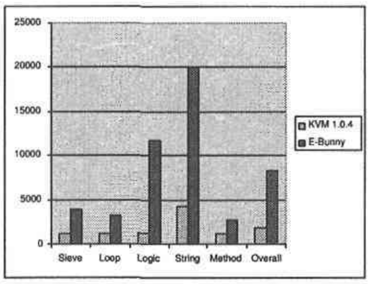


Figure 4.15: CaffeineMark scores of KVM and E-Bunny with GCC

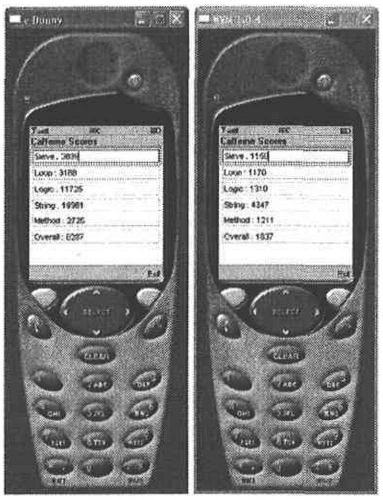
4.4 Implementation and Results

E-Bunny is implemented using the C programming language. In our experiments, we used the GNU *C* compiler (GCC) to build the KVM (version 1.0.4) with E-Bunny. Table 4.1 shows the executable size of KVM with and without E-Bunny. The first column gives the total executable footprint of KVM without E-Bunny. The second column gives the total executable footprint of KVM equipped with E-Bunny dynamic compiler. Finally, column 3 of the table shows that using GCC to build KVM with E-Bunny produces a footprint overhead of 64 KB. To summarize, E-Bunny requires 64 KB for executable footprint overhead, 64 KB for storing translated methods and 10 KB for the map between bytecodes and native instructions, which is used for compiling control flow instructions and for exception handling mechanism. Hence, the total memory resources required by E-Bunny is 138 KB.

To evaluate the performance of E-Bunny, we have run CaffeineMark benchmark (without the float test) on the original version of KVM with and without E-Bunny.

E-Bunny produces an overall speedup of 400% over the original KVM (Compare overall score in Figure 4.15). Note that a higher score, in this Figure, means better performance. Particularly, the *String* test is drastically improved thanks to our dynamic compiler since this test contains a loop in which a method that appends strings is frequently called. Moreover, we built the MIDP 2.0 profile, intended to CLDC devices, using E-Bunny and we ran successfully several midlets. Figure 4.16 shows a snapshot of MIDP emulator illustrating CaffeineMark midlet results.

Figure 4.17 outlines an example of an output for the method *execute* of the Sieve test, which is one of CaffeineMark test suites.



```
Caffeine Scores
Sieve : 3839
Loop : 3188
Logic : 11725
String : 19981
Method : 2726
Overall : 6287
```

```
Caffeine Scores
Sieve : 1150
Loop : 1170
Logic : 1310
String : 4347
Method : 1211
Overall : 1837
```

Figure 4.16: CaffeineMark midlet ran by KVM and by E-Bunny

4.5 Conclusion

We reported, a new acceleration technology for Java embedded virtual machines that is based on dynamic selective compilation. This technology targets the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) platform. We designed and implemented an efficient, lightweight and low-footprint dynamic compiler for embedded Java Virtual Machine. We presented the architecture, the design as well as the technical issues of E-Bunny and how we addressed them. Experimental results demonstrated that we accomplished a speedup of 400% with respect to the Sun's latest version of KVM.

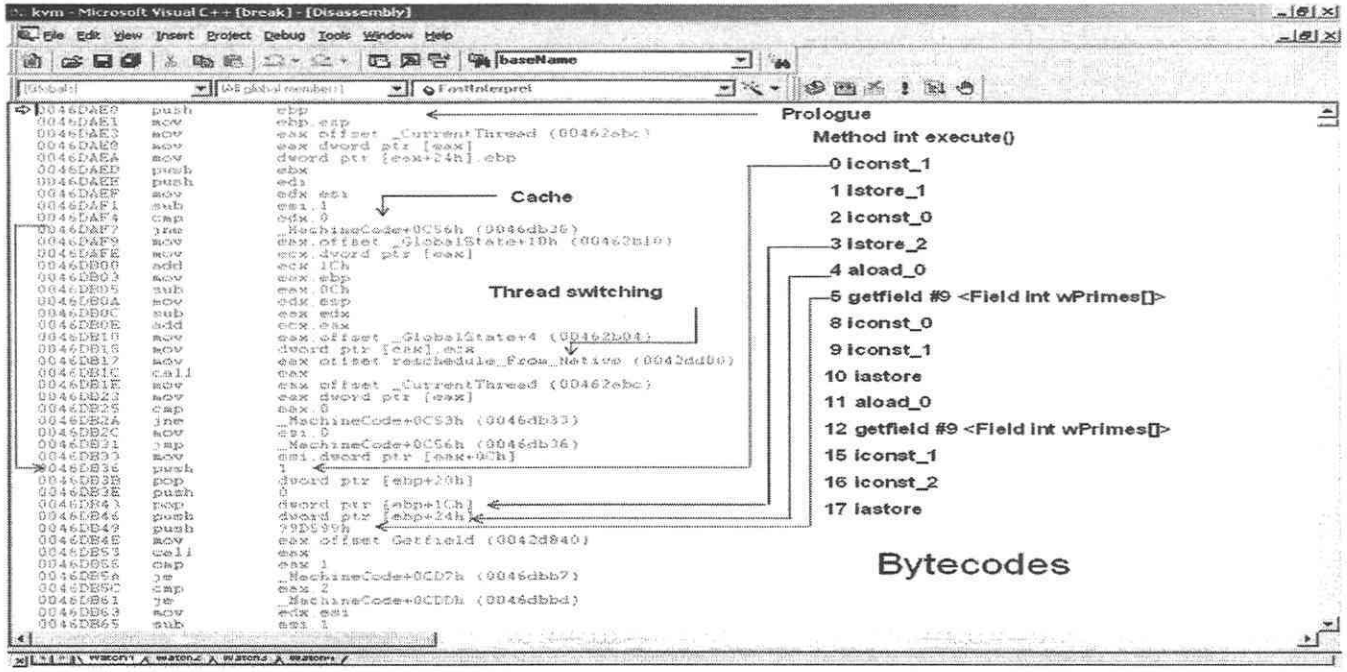
Many enhancements can be introduced in E-Bunny. The major one concerns the bidirectional smooth switching between the interpreted and compiled modes. In fact, the profiling strategy adopted in the current version of E-Bunny, which consists of compiling each method called from the compiled one, is less complex to implement. However, it presents a drawback since it leads to compile non-performance-critical methods. On the other hand, a more efficient centralized thread scheduling is needed. This is expected to reduce the generated native code size.

In the next chapter, we propose two dynamic acceleration techniques that can work together with an embedded dynamic compiler like E-Bunny:

- The first is an acceleration technique that is related to the generation of a native interpreter and which works together with E-Bunny. This technique allows to avoid the drawback of switching between two stacks (Java and native ones) by the design of a unified stack model.

- The second is a hash-based technique for accelerating the dynamic method lookup mechanism. This technique is dynamic, flexible and efficient.

These two techniques are designed in the scope of a synergy between a fast interpretation and lightweight dynamic compilation.


kvm - Microsoft Visual C++ [break] - [Disassembly]
File Edit View Insert Project Debug Tools Window Help
baseName
(Globals)
(All global members)
FastInterpret
Prologue
Cache
Thread switching
Watch1
Watch2
Watch3
Watch4

```
0046DAE0   push    ebp
0046DAE1   mov     ebp,esp
0046DAE3   mov     eax,offset _CurrentThread (00462ebc)
0046DAE8   mov     eax,dword ptr [eax]
0046DAEA   mov     dword ptr [eax+24h],ebp
0046DAED   push    ebx
0046DAEE   push    edi
0046DAEF   mov     edx,esi
0046DAF1   sub     esi,1
0046DAF4   cmp     edx,0
0046DAF7   jne     _MachineCode+0C56h (0046db36)
0046DAF9   mov     eax,offset _GlobalState+10h (00462b10)
0046DAFE   mov     ecx,dword ptr [eax]
0046DB00   add     ecx,1Ch
0046DB03   mov     eax,ebp
0046DB05   sub     eax,0Ch
0046DB0A   mov     edx,esp
0046DB0C   sub     eax,edx
0046DB0E   add     ecx,eax
0046DB10   mov     eax,offset _GlobalState+4 (00462b04)
0046DB15   mov     dword ptr [eax],ecx
0046DB17   mov     eax,offset reschedule_From_Native (0042dd80)
0046DB1C   call    eax
0046DB1E   mov     eax,offset _CurrentThread (00462ebc)
0046DB23   mov     eax,dword ptr [eax]
0046DB25   cmp     eax,0
0046DB2A   jne     _MachineCode+0C53h (0046db33)
0046DB2C   mov     esi,0
0046DB31   jmp     _MachineCode+0C56h (0046db36)
0046DB33   mov     esi,dword ptr [eax+0Ch]
0046DB36   push    1
0046DB3B   pop     dword ptr [ebp+20h]
0046DB3E   push    0
0046DB43   pop     dword ptr [ebp+1Ch]
0046DB46   push    dword ptr [ebp+24h]
0046DB49   push    99D599h
0046DB4E   mov     eax,offset Getfield (0042d840)
0046DB53   call    eax
0046DB55   cmp     eax,1
0046DB5A   je      _MachineCode+0CD7h (0046dbb7)
0046DB5C   cmp     eax,2
0046DB61   je      _MachineCode+0CDDh (0046dbbd)
0046DB63   mov     edx,esi
0046DB65   sub     esi,1
```

Method int execute()

```
0 iconst_1
1 istore_1
2 iconst_0
3 istore_2
4 aload_0
5 getfield #9 <Field int wPrimes[]>
8 iconst_0
9 iconst_1
10 iastore
11 aload_0
12 getfield #9 <Field int wPrimes[]>
15 iconst_1
16 iconst_2
17 iastore
```

Bytecodes

Figure 4.17: Snapshot of an E-Bunny generated code

Chapter 5

A Synergy Between Efficient Interpretation and Dynamic Compilation

5.1 Introduction

Strengthened by the results of the E-Bunny project and the downstream insights, we started looking for more acceleration. This led us to the idea of establishing a synergy between efficient interpretation and dynamic selective compilation.

In this chapter, we propose two acceleration techniques for embedded Java platforms: (1) the use of a threaded interpreter that collaborates with a fast one-pass dynamic selective compiler to translate performance-critical methods to native code; (2) the acceleration of the method lookup mechanism by a hashing technique.

The first acceleration technique advocates the use of a threaded interpreter and a fast one-pass dynamic selective compiler, like E-Bunny, to translate performance-critical methods to native code. The threaded interpreter is a pool of codelets. Each codelet carries out the dispatch to the next bytecode eliminating therefore the need for a costly centralized traditional dispatch mechanism. The translation process takes advantage of the threaded interpreter by reusing most of the previously mentioned codelets. This tight collaboration between the interpreter and the dynamic compiler leads to a fast and lightweight (in terms of footprint) execution of Java class files.

The second technique relies on improving the method lookup that is implemented in the KVM. This technique is dynamic, flexible and efficient. In fact, it is performed at the loading-time. Moreover, it offers the possibility to developers of embedded VMs for making a tradeoff between the memory space and execution time. Finally, this technique proved its efficiency w.r.t the KVM original method lookup.

To sum-up, the main contributions of our acceleration techniques are the following:

- The design of a generated threaded native interpreter. This is defined as a pool of codelets. An important subset of these codelets is suitable for code reuse.
- A fast one-pass dynamic selective compiler based on the reuse of interpreter codelets.
- A smooth switching mechanism between the interpreted and compiled modes.
- A fast method lookup mechanism based on the cooperation of two components of the KVM: the loader and the interpreter. The loader builds a method hashing table and the interpreter uses it to perform a quick method lookup.

5.2 Compiling by Code Reuse

Accelerating the interpretation mechanism has been and is still a focus of interest for many researchers. Generally a pure bytecode interpreter is an infinite loop embedding a switch-case statement that dispatches to a sequence of bytecodes. Each switch-case value implements one Java bytecode. This entails a significant overhead. To circumvent this drawback, the use of direct threaded interpretation has been suggested. The latter is an interpretation technique introduced in the Forth programming language [45]. Thanks to this technique, the central dispatch is eliminated. Each bytecode of the method being interpreted is replaced by an address of a corresponding implementation. In addition, such an implementation ends with the required dispatch to the next opcode.

The inline threading interpretation technique [101] improves upon the direct threading technique by eliminating the dispatch overhead within basic blocks. The former technique identifies bytecode sequences that form basic blocks. A new implementation is then dynamically created for such sequences by copying and catenating each bytecode's implementation in a new buffer. The dispatch code is then copied at the end.

Hendren et al. [48] proposed a technique that copes with the difficulties that arise when adapting the inline threading technique to Java. In fact, Java's features such as lazy class initialization, lazy class loading/linking and multi-threading support conflict with the implementation of inline-threaded interpretation technique. A technique called "preparation sequence" has been proposed in [48] to solve the problems caused by in-place code replacement within an inline-threaded interpreter of a Java virtual machine.

These acceleration techniques of the interpretation mechanism achieve a reasonable speed-up. However, they fail to compete with those approaches that introduce some sort of compilation (AOT, JIT, dynamic selective compilation, etc.) [4, 10, 25, 60, 78, 124, 139].

The closest work to our proposed technique can be found in [75]. The authors presented a technique that combines interpretation with compilation to get a sort of hybrid interpretation strategy. This technique targets embedded systems and presents a code generation technique that leverages the interpreter self-code. The interpreter is written in the C language to achieve portability. This limits however the interpreter code reuse. Moreover, the compilation targets method fragments. The rationale underlying this choice is to reduce the size of the generated code. However, by doing so, the switching frequency between the interpreter and the compiler modes is increased and therefore an additional complexity and overhead are introduced. The method advanced by the authors requires: (1) the use of some kind of bytecode analysis to detect basic blocks; (2) the use of a peephole optimizer to improve the quality of the produced code. These two analyses lead to an additional overhead.

Our interpretation technique leverages also the interpreter code but it deviates from [75] in two directions:

- First, the interpreter code is efficiently generated and is not the result of a high-level compiler. This makes it suitable for reuse by a dynamic compiler to generate code by copy and concatenation.
- Second, the compilation unit in our technique is a method. By doing so, we reduce significantly the technical complexity of the switching mechanism and the underlying overhead and frequency.

The threaded interpretation achieves a relative speed-up over the switch-case based interpretation [48, 101]. In order to reach significant performance, it is necessary to introduce dynamic compilation. Embedding a dynamic compiler into a Java virtual machine targeting small devices limits severely the application of traditional optimizations (flow-based analysis). In this context, dynamic selective compilation is the most adequate approach. Hotspot methods should be dynamically identified and compiled at run-time. The compilation cost in time and footprint should be lightweight.

It is crucial to design a system allowing a cooperation between an efficient interpretation mechanism together with a lightweight dynamic selective compilation. The technique fits in this framework. The main idea is to generate a native threaded interpreter. This is pool of code units, called codelets. Each codelet is a native implementation of a Java bytecode. This interpreter is generated at the start-up of the Java virtual machine or better offline if the target architecture is known. When a method is identified as hotspot, it is dynamically compiled. The compiler makes use of the interpreter codelets during the code generation process. This leads to a lightweight compilation mechanism that is appropriate in an embedded context.

Generating a native interpreter allows us to implement the Java stack on the native stack. This induces a fast switching between the interpreter and compiler modes of execution since the frames of an interpreted method and a compiled one are on the same stack unlike what is proposed in the previous chapter. The transfer of parameters between the Java and the native stack is no more needed when switching from the interpreter mode to the native mode and vice versa. Our technique is sustained by a smooth and uniform switching mechanism.

5.2.1 Generated Native Threaded Interpreter

Since the dynamic compiler reuses the codelets during the code generation process, these codelets have to be implemented in a way that facilitates code reuse.

We distinguish two categories of bytecodes: context-free bytecodes and context-dependent bytecodes. They are defined as follows:

- A context-free bytecode can be translated to a native code that is independent of dynamic information. For instance, the bytecode `aload_0` is always translated to a `push` instruction of the 0^{th} local variable into the stack.
- A context-dependent bytecode requires some dynamic information (e.g. instruction pointer) to be translated to native code. For instance, the bytecode `iload`

nop	aconst_null	iconst_m1	iconst_0
iconst_1	iconst_2	iconst_3	iconst_4
iconst_5	lconst_0	lconst_1	iload_0
iload_1	iload_2	iload_3	lload_0
lload_1	lload_2	lload_3	aload_0
aload_1	aload_2	aload_3	istore_0
istore_1	istore_2	istore_3	lstore_0
lstore_1	lstore_2	lstore_3	astore_0
astore_1	astore_2	astore_3	pop
pop2	dup	dup_x1	dup_x2
dup2	dup2_x1	dup2_x2	swap
iadd	ladd	isub	lsub
imul	lmul	idiv	ldiv
irem	lrem	ineg	lneg
ishl	lshl	ishr	lshr
iushr	lushr	iand	land
ior	lor	ixor	lxor
i2l	l2i	i2b	i2c
i2s	lcmp	return	ireturn
areturn	lreturn	arraylength	iaload
laload	iastore	lastore	baload
bastore	castore	caload	saload
sastore	aaload	aastore	monitorenter
monitorexit	athrow	customcode	

Table 5.1: Context-free bytecodes

requires computing the index of a local variable, which is extracted from the bytecode stream using the instruction pointer.

A scan of the instruction set (without floats) of the Kilo Virtual Machine (KVM) proves that 95 bytecodes are context-free while the number of context-dependent bytecodes is 66. Hence, context-free bytecodes represent more than 59% of the standard bytecodes handled by the KVM. The set of the context-free bytecodes is presented in Table 5.1.

The remaining bytecodes fit in the context-dependent category and are presented in Table 5.2.

The main motivation underlying our bytecode taxonomy is to confine the codelet reuse to context-free bytecodes. Actually, these bytecodes are very frequently used in Java applications as exemplified by [110]. According to the results of [110], it is easy to see that our context-free bytecodes correspond to the Loads, Stores and ALU categories, which are very frequently used. For instance, with respect to the SPECjvm98 bench-

bipush	sipush	ldc	ldc_w
ldc2_w	iload	lload	aload
istore	lstore	astore	iinc
ifeq	ifne	iflt	ifge
ifgt	ifle	if_icmpeq	if_icmpne
if_icmplt	if_icmpge	if_icmpgt	if_icmple
if_acmpeq	if_acmpne	goto	tableswitch
lookupswitch	newarray	anewarray	multianewarray
ifnull	ifnonnull	goto_w	new
instanceof	checkcast	wide	putfield
getfield	getstatic	putstatic	invokevirtual
invokespecial	invokestatic	invokeinterface	getfield_fast
getfieldp_fast	getfield2_fast	putfield_fast	putfield2_fast
getstatic_fast	getstaticp_fast	getstatic2_fast	putstatic_fast
putstatic2_fast	invokevirtual_fast	invokespecial_fast	invokestatic_fast
invokeinterface_fast	new_fast	anewarray_fast	multianewarray_fast
checkcast_fast	instanceof_fast		

Table 5.2: Context-dependent bytecodes

mark [30], these categories represent respectively 35.54%, 6.65% and 9.94% of the total bytecodes of this benchmark. Hence, the compilation process of performance-critical methods by copying interpreter codelets instead of regenerating them is improved. This relies on the relatively high frequency of context-free bytecodes. In the sequel, we highlight the interpreter structure and components.

The present technique is based on a generated native threaded interpreter. The generation of the interpreter is one-time virtual machine operation performed at the start-up. Basically, the interpreter may be considered as a pool of code units called codelets. Each codelet is an efficient native implementation of a bytecode. The implementation of this interpreter uses a data structure composed of a jump table and a codelet table. Each entry in the codelet table contains the generated native implementation of the corresponding bytecode.

The interpretation main loop is reduced to a jump into the codelet table via the jump table according to the current bytecode in the method under interpretation. In addition, each codelet ends up by dispatching to the next bytecode. This eliminates the traditional centralized bytecode dispatch overhead. Besides, the codelets are generated in a way that allows to reuse them in the code generation phase during a method compilation. Figure 5.1 outlines the interpreter architecture.

The template of a codelet for a bytecode is as follows:

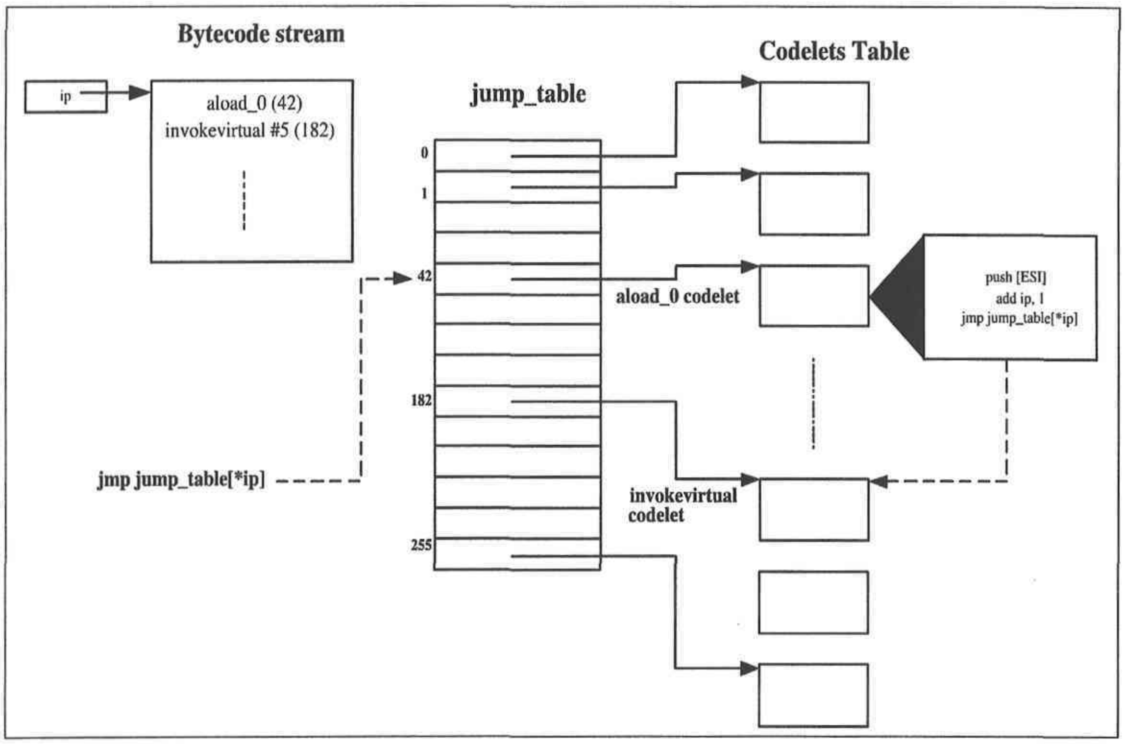


Figure 5.1: Generated native threaded interpreter structure

```
push the 3rd local variable
add ip, 1
jmp jump_table[*ip]
```

Figure 5.2: Codelet for `aload_3`

- Native code;
- Incrementation of the instruction pointer (ip);
- Jump to the entry corresponding the bytecode at the current ip.

Figure 5.2 illustrates the codelet, in assembly-like pseudo-code, for the `aload_3` bytecode.

Figure 5.3 depicts the algorithm, in pseudo-code, for the interpreter codelets generation. The start-up of the interpretation of a method is reduced to a jump to the codelet associated with the opcode of the first bytecode of this method. The codelets are generated and their respective addresses are stored in a jump table. The latter maps each opcode with the address of the corresponding codelet.

```
generateInterpreter(ip) {
  generate a jump instruction to jump_table[*ip]
  for each bytecode
    generate the corresponding native code
    insert the codelet address into the next entry of jump_table
}
```

Figure 5.3: Interpreter codelet generation

5.2.2 Reusing Codelets for Dynamic Compilation

Dynamic compilation occurs at run-time. The compilation overhead is then a critical issue particulary in the embedded context. Therefore, it is important to minimize the compilation time in order to reduce the overall execution time. The classical compilation techniques (flow analysis, aggressive optimizations etc.) produce a high code quality. They require however, huge data structures and consume time, which makes them unaffordable when targeting virtual machines that are meant to be embedded in resources-constrained devices.

The code reuse technique that we propose is a natural continuation of the work done on E-Bunny. Thanks to this technique, we avoid the systematic regeneration of native code each time a performance-critical method is detected. We achieve this goal by the reuse of already-generated code for the interpreter at the virtual machine start-up (or offline).

Context-free bytecodes are translated by a simple copy of the already generated codelets. Thus, we save the time spent in regenerating it. However, to avoid recomputing dynamic information such as instruction pointer values, constant values in constant pools, as it is required by the interpretation process, the dynamic compiler computes them efficiently, once for all, for the method under compilation. Hence, context-dependent bytecodes are translated to native code using these dynamic information. Figure 5.4 depicts the native code generation scheme used in this technique.

Figure 5.5 outlines the algorithm, in pseudo-code, for compiling a bytecode. When the opcode of the bytecode under compilation is context-free, the interpreter codelet associated with this opcode is just copied and concatenated to the already generated code in a buffer. A special care is taken to discard the original dispatch code in each codelet. In the case of a context-dependent bytecode, the compiler generates an efficient code from scratch. The compiler uses the dynamic information available at run-time, such as the value of *ip*, to generate efficient native code. Section 5.2.4 gives more details on this issue.

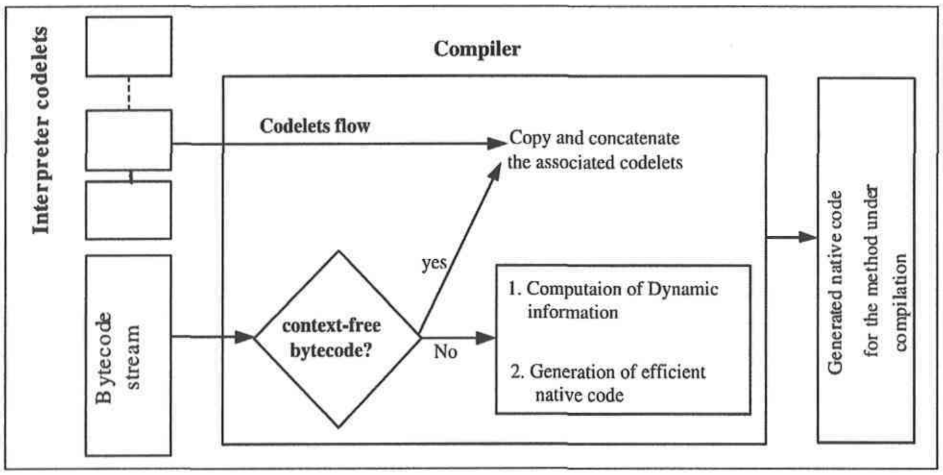

Interpreter codelets
Compiler
Codelets flow
Copy and concatenate the associated codelets
yes
Bytecode stream
context-free bytecode?
No
1. Computaion of Dynamic information
2. Generation of efficient native code
Generated native code for the method under compilation

Figure 5.4: Native code generation scheme

```
compile(bytecode) {
  if is_context_free(bytecode)
    copy jump_table[bytecode]
  else {
    compute dynamic information
    generate native code for the bytecode
  }
}
```

Figure 5.5: Compilation of a bytecode

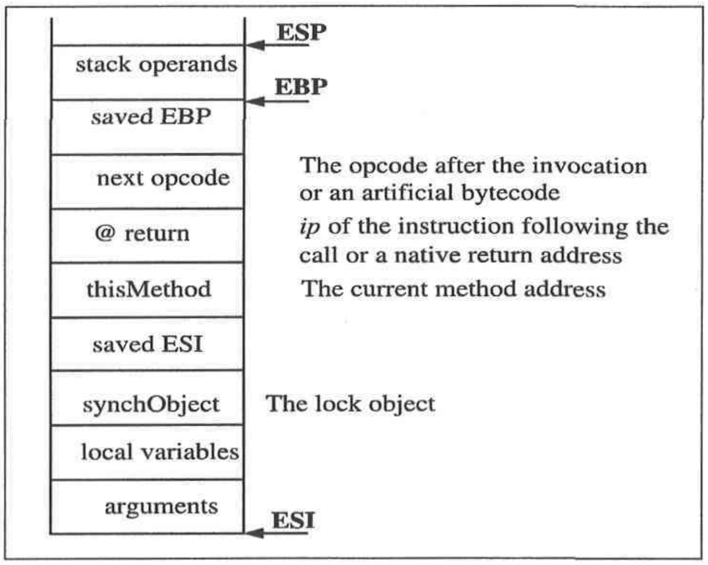

ESP
EBP
ESI
stack operands
saved EBP
next opcode
The opcode after the invocation or an artificial bytecode
@ return
ip of the instruction following the call or a native return address
thisMethod
The current method address
saved ESI
synchObject
The lock object
local variables
arguments

Figure 5.6: The unified stack layout

5.2.3 Smooth Switching Mechanism

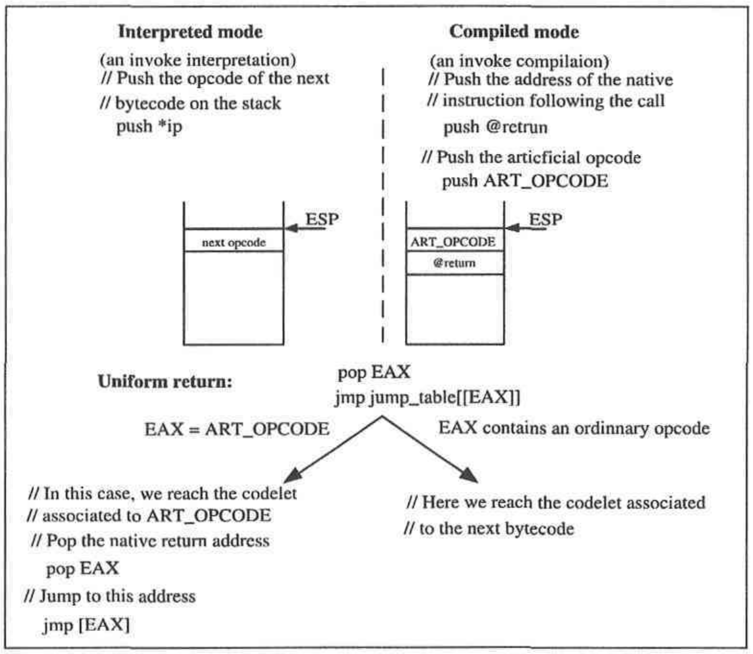
The technique, we propose, is supported by a smooth and uniform switching mechanism between the interpreted and compiled modes. In order to achieve this goal, we use a unique native stack per thread. In the sequel, we illustrate the stack layout that supports the switching mechanism and describe the switching implementation.

A straightforward translation approach would maintain a run-time stack and manipulates it the same way the interpreter does with the Java stack [60]. Consequently, the bytecode execution is based on two stacks (Java stack and native stack). The switching between the interpreter and native modes is much expensive. This is due to the unnecessary memory traffic between the two stacks.

We propose a design where the frames of an interpreted method and a compiled method for a thread are represented in the same native stack. This leads to a smooth and fast switching from the interpreted to the compiled mode and vice versa. Indeed, using a unique execution stack (native stack) saves the overhead induced by transfers of call parameters from the Java stack to the native stack. Figure 5.6 depicts the unified stack layout.

Besides, some registers are dedicated to hold some information. For instance, the ESI register contains arguments address whereas the EBX register contains the instruction pointer (*ip*).

The use of a unique stack speeds up the switching between interpreted and compiled methods. However, it introduces new issues. In fact, we need to know the kind of the



```
(an invoke interpretation)
// Push the opcode of the next
// bytecode on the stack
  push *ip
```

```
(an invoke compilaion)
// Push the address of the native
// instruction following the call
  push @retrun
// Push the articficial opcode
  push ART_OPCODE
```

```
pop EAX
jmp jump_table[[EAX]]
```

```
// In this case, we reach the codelet
// associated to ART_OPCODE
// Pop the native return address
  pop EAX
// Jump to this address
  jmp [EAX]
```

```
// Here we reach the codelet associated
// to the next bytecode
```

Figure 5.7: A lightweight interpreter/compiler mode switching mechanism

calling method (interpreted or compiled) to restore the calling context.

Actually, returning to an interpreted method resumes the execution at the next opcode following the invocation bytecode whereas returning to a compiled method resumes the execution at the next native instruction following the call. Hence, the return address has different semantics in our design depending on the calling method type.

We propose a uniform mechanism allowing to restore smoothly the calling method context. This is based on the use of an artificial opcode and a specific codelet associated with it. When interpreting a method invocation bytecode (invokevirtual, invokespecial, invokeinterface, invokestatic etc.), the opcode of the following bytecode is pushed into the stack. In the case of a compiled method, the native return address and the artificial opcode are pushed into the stack. This artificial opcode allows to restore the context of the compiled calling method transparently without any explicit test. Actually, the codelet associated with this artificial opcode is responsible for jumping to the native address previously pushed into the stack as well. When returning to an interpreted method, a jump to the codelet associated with the already pushed opcode is performed. Figure 5.7 outlines the implementation of this lightweight uniform switching mechanism.

The switching is implemented by means of two stubs. A prologue stub ensures saving the context of a calling method and setting the context of the called one. Reciprocally, an epilogue stub is responsible for restoring the context of the calling method. Figures

```
prologue() {
  //leave space for n local variables
  sub ESP, n*4
  //push the call receiver object into the stack
  push object
  push ESI
  //push the method pointer
  push thisMethod
  //push return address (ip or pc)
  push returnAddress
  // artificial opcode in case of compiled
  // or opcode of the next instruction
  push opcode
  push EBP
  mov EBP, ESP
}
```

Figure 5.8: Prologue

5.8 and 5.9 outline these stubs.

5.2.4 Scenario

We illustrate, in the present section, the code generation technique based on the interpreter codelets reuse to compile Java bytecodes. Figure 5.10 illustrates the bytecode sequence under compilation.

Figure 5.11 illustrates the code generation technique in action. The righthand part of Figure 5.11 depicts a snippet of the interpreter codelets generated at the virtual machine start-up whereas the left-hand part depicts how the different bytecodes are handled.

The bytecodes `aload_1 (43)` and `iadd (96)` belong to the context-free category mentioned above. The native code generation for these bytecodes is then reduced to a copy of the corresponding interpreter codelet as depicted in Figure 5.11.

The bytecode bipush 5, is however a context-dependent bytecode. Indeed, an efficient compilation of this bytecode requires the actual current value of *ip* in order to access the index associated with the opcode `bipush` in the bytecode stream (method), which is 5 in the present example. This information is not available at the generation of the interpreter. We recall that the interpreter generation is a one-time virtual machine operation, which occurs prior to the execution of any Java method. As a result, the codelet associated with `bipush` could not be reused.

```
epilogue( ) {
  pop EBP
  //artificial opcode or opcode of the next instruction
  pop opcode
  //return address (ip or pc)
  pop EBX
  //remove the method pointer from the stack
add ESP, 4
pop ESI
//remove the call receiver object from the stack
add ESP, 4
mov ESP, ESI
jmp jump_table[opcode]
}
```

Figure 5.9: Epilogue

```
aload_1
bipush 5
iadd
```

Figure 5.10: Bytecode sequence under compilation

The dynamic compiler (like E-Bunny), relying on the dynamic information available at run-time (value of *ip* for instance), generates an efficient code for context-dependent bytecodes from scratch. The value to push (which is 5) is extracted from the bytecode stream. The generated code is just a push of the latter value into the stack: push 5. The interpreter codelet is inefficient because it contains the extra instructions required to access the value in the bytecode then afterwards it has to push this value into the stack.

5.3 Enhancing Interpretation by Method Call Acceleration

The second acceleration technique we propose is related to another important mechanism which is the method lookup. The acceleration of this feature is needed since the dynamic compilation enhances just the execution of a method while the method lookup mechanism remains the same. In the sequel, we give a detailed idea about the method lookup mechanism.

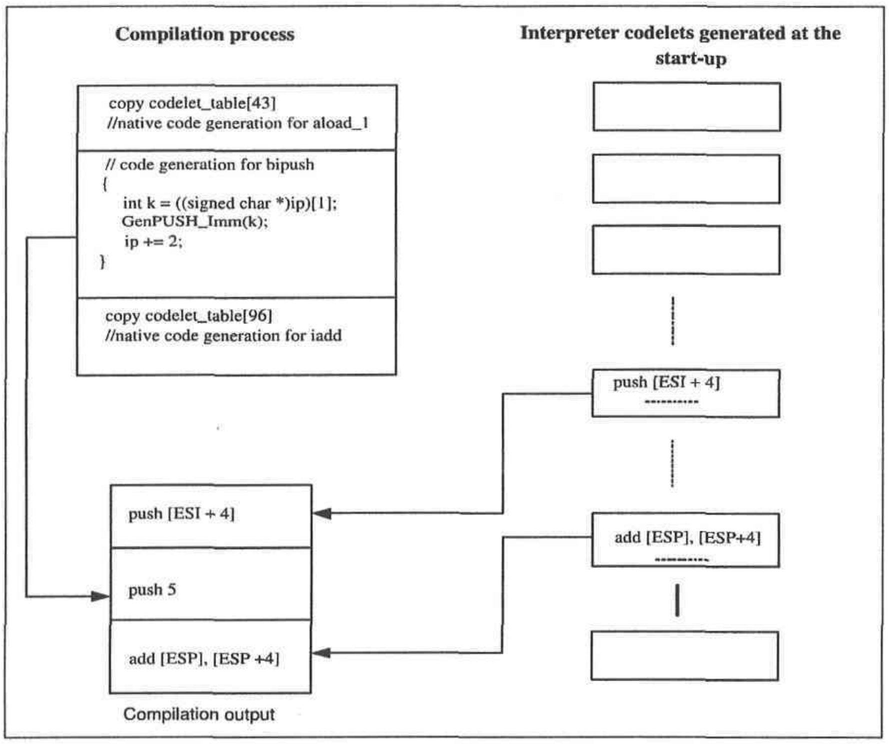


Figure 5.11: A scenario of the proposed compilation by code reuse technique

5.3.1 Method Lookup Mechanism

Object-oriented languages support inheritance and polymorphism to allow the development of flexible and reusable software. The type of a specific object would usually be determined at run time. This feature is called the dynamic binding. In this context, the selection of the appropriate method to execute is based on a lookup mechanism, which means that the actual method to be executed after an invocation is determined dynamically using the type of the method's receiver, the class hierarchy and the method inheritance schema. The lookup mechanism consists of determining the actual method to be executed when an invocation occurs. If this class implements a method that has the same signature (name and parameter types) as the called one, the found method will be executed. Otherwise, the parent classes will be checked recursively until the searched method is found. If no method is found, an error is signaled (*MsgNotUnderstood*). Unfortunately, this operation is too frequent and is very expensive.

The principal dynamic binding algorithm is called the Dispatch Table Search [54] (DTS). It proceeds as mentioned above. The DTS is good in terms of memory cost, however the search overhead makes the mechanism too slow. Many techniques were proposed to minimize the overhead associated to DTS: static techniques which precompute a part of the lookup and dynamic techniques which cache the results of previous lookup, thus avoiding other lookups.

The Selector Table Indexing technique (STI) [31] is one of the static techniques aiming to enhance the method lookup mechanism. It operates as follows. Given a class hierarchy of C classes and S selectors (method names), a two-dimensional table of $C * S$ entries is built. Classes and selectors are given consecutive numbers and the table is filled by pre-computing the lookup for each class and selector. A table entry contains a reference to the corresponding method or to an error routine. These tables are computed for a complete system. The main drawback of STI is that space requirements are huge for a large system. Hence, many dispatch table compression techniques were proposed (Selector coloring [39], Row displacement [42], etc.) to minimize space overhead. Another drawback of this technique is that the computed table is very sensitive to changes in the class hierarchy. However, this technique delivers fast and constant time lookup.

The devirtualization technique with code patching mechanism [64] is another optimization technique that converts a virtual method call to a direct call. Given a method call, the current class hierarchy is analyzed by the compiler to determine if the call can be devirtualized. If it is true and if the method size is small, the compiler generates the inlined code of the method with the backup code of making the dynamic call. When devirtualization becomes invalid, the compiler performs code patching to make the backup code executed later. Otherwise (devirtualization is valid), only the inlined code is actually executed. The main drawback of this technique is that it relies on heavy analysis (flow-sensitive type analysis, dynamic class hierarchy analysis, etc.) so it is not convenient for embedded systems due to the fact that it can be too expensive in both time and space.

A static method call resolution technique [131] was proposed to solve dynamic method calls. A variable-type analysis and a declared-type analysis use the whole class hierarchy program to compute a set of method call receivers. This technique is limited because it does not deal with the dynamic class loading problem. In fact, the class hierarchy could change while the program is executing. This could change a method call receivers set and make the static performed optimizations inaccurate.

Dynamic techniques consist of caching results of previous lookups. Cache-based techniques eliminate the requirements to create huge dispatch tables, so memory overhead and table creation time are reduced. There are two main approaches to caching: global caches [54] and small inline caches [40]. The global cache technique stores the previous lookup results. In the global cache table, each entry consists of triplets (receiver class, selector and method address). The receiver class and the selector are used to compute an index in the cache. If the current class and the method name match those in the cached entry at the computed index, the code at the method address is executed. Hence, method lookup is avoided. Otherwise, a default dispatching technique (usually DTS) is used and at the end of this search, a new triplet is added to the cache table and control is transferred to the found method. The run-time memory required by this algorithm is small, usually a fixed amount of the cache and the overhead of the DTS technique. The main disadvantage of this technique is that a frequent change of the receiver class slows the execution.

The inline cache technique consists of caching the result of the previous lookup (method address) in the code itself at each call site. Inline cache changes the call instruction by overwriting it with a direct invocation of the method found by the default method lookup. Inline cache assumes that the receiver's class changes infrequently, otherwise, the inline cache technique delivers slow execution time.

The polymorphic inline cache [59] is an extension of the inline cache technique. The compiler generates a call to a special stub routine. Each call site is a jump to a specific stub function. The function is initially a call to a method lookup. Each time the method lookup is called, the stub function is extended. In each extension, to execute a method code a test on method and class names is added to the stub function. This technique has the cost of a test and a direct jump in the best case. Moreover, the executable code could expand dramatically when the class hierarchy is huge and the receiver class changes frequently.

Hereafter, we propose a dynamic, flexible and efficient technique for accelerating the method lookup mechanism in embedded Java virtual machines.

5.3.2 Method Lookup Acceleration

The method lookup is accelerated by the application of a direct access to method tables. This is achieved by using an appropriate hashing technique. Actually, we build a hash table for each virtual method table at the loading-time. Each index of the hash table is a hashing result of the method signature. The size of the hash table should be carefully chosen so as to have a low footprint while minimizing the collisions between method signatures. By doing so, we get a more efficient and flexible lookup mechanism. Efficiency stems from the fact that we have direct access to method tables. Flexibility stems from the fact that we can tune the size of the hash table so as to have the best ratio for speed versus footprint. In what follows, we explain how this lookup acceleration could be implemented within the standard embedded Java virtual machine such as KVM [81] (Kilobyte Virtual Machine). The method lookup mechanism in KVM is linear i.e. it uses a sequential access. A hash-based lookup will definitely yield a better performance. The implementation of such a mechanism will affect two components of the virtual machine: the loader and the interpreter. The loader is

```
buildMethodHashTable() {
 for each method of the class method table {
  h = compute_hash(method);
  element = get_element(HashTable,h);
  if (element->flag == ON) {
   allocate_space(method);
   register_method_in_collision_list();
  }
  else {
   register_method_in_HashTable();
   method->flag = ON;
  }
 }
}
```

Figure 5.12: Method hash table construction algorithm

modified to construct hashed method tables for each loaded class. The interpreter is modified to take advantage of the new method tables to perform fast and direct-access lookups. During the loading process of a class, a hash method table is built. A hash is computed from the method signature. Each entry in the hash table consists of two components:

- The first component is a flag indicating whether the class contains a method with such a definition.
- The second component is a pointer to the method definition. In the case of a collision, this second component is a pointer to a list of method definitions.

The method hash table construction algorithm is depicted in Figure 5.12.

The original lookup algorithm is linear. It tests in each method table of a class, by iteration over its elements, if it has a method that has the same signature as that invoked (key). In Figure 5.13 we give the original lookup algorithm.

The new lookup algorithm uses the hash obtained from the method signature to access the corresponding entry in the hash table. If the flag associated with this entry is ON, it accesses the method definition thanks to the second component of the entry. If the flag is OFF, this means that the class does not implement such a method and the search is directed to the super-class. In Figure 5.14, we give the new lookup algorithm.

The new lookup algorithm performs fewer iterations than the original one. In fact, in the worst case, the whole collision list has to be visited. The lookup method acceleration

```
lookupMethod(class,key) {
  while (class) {
    table = get_method_table(class);
    for each method of table {
      if (method->signature == key) {
        return method;
      }
    class = class->superclass;
    }
  }
}
```

Figure 5.13: Original lookup algorithm

```
lookupMethod(class,key) {
  while (class) {
    HashTable = get_HashTable(class);
    h = compute_method_hash(key);
    entry = get_element(HashTable,h);
    if (entry->flag == ON) {
      for each element in collision list {
        if (key == element->signature)
          return element;
      }
    }
    class = class->superclass;
  }
}
```

Figure 5.14: Optimized lookup algorithm

depends on the hash table size. In fact, a big size requires a high memory space but it minimizes the collisions. On the other hand, it might induce an additional cost in terms of memory management (allocation, garbage collection, compaction, etc.).

5.3.3 Caching

Another technique for the acceleration of a method call is caching. We claim that the traditional inline cache technique, described previously, could achieve a significant speed-up of a program execution by slightly modifying the cache layout. Actually, the modification consists of adding a pointer to the receiver object in the cache entry. We explain hereafter why such a simple modification will result in a speed-up. In the conventional inline cache technique (such as the one implemented in KVM), only a pointer to the method definition is stored in the cache structure. When a method is invoked, the class of the receiver is compared to the class of the invoked method. If there is an equality between these two classes, the method definition is retrieved thanks to the cache entry. If there is no equality, a dynamic lookup is performed to search for the method definition in the class hierarchy. This inline cache technique could be significantly improved by avoiding many of the dynamic lookups when there is a mismatch between the class of the receiver and the class of the invoked method. Actually, when there is such a mismatch, if we can detect that the receiver has not changed, we can retrieve the method definition from the cache. This is done by:

- Adding a pointer to the receiver in the cache structure,
- Modifying the condition that guards cache retrieval. Actually, when a method is invoked, the condition to get a method definition from the cache is:
  - The class of the receiver is equal to the class of the invoked method, or,
  - The current receiver is equal to the cached receiver (the receiver has not changed).

Here is an illustration when this inline cache technique yields a significant speed-up. Assume that we have a class B that inherits a non-static method m from a class A. Assume also that we have a loop that will be executed very often in which we have the following method invocation: the method m is invoked on an object say o_B that is instance of the class B. In our inline caching technique the object o_B is going to be cached after the first invocation of the method m. In the subsequent invocations of the method m, since the class of the receiver (B) is different from the class of the invoked method (A), the behavior of the conventional inline cache technique will be very different from the one of the proposed inline technique. The conventional technique will

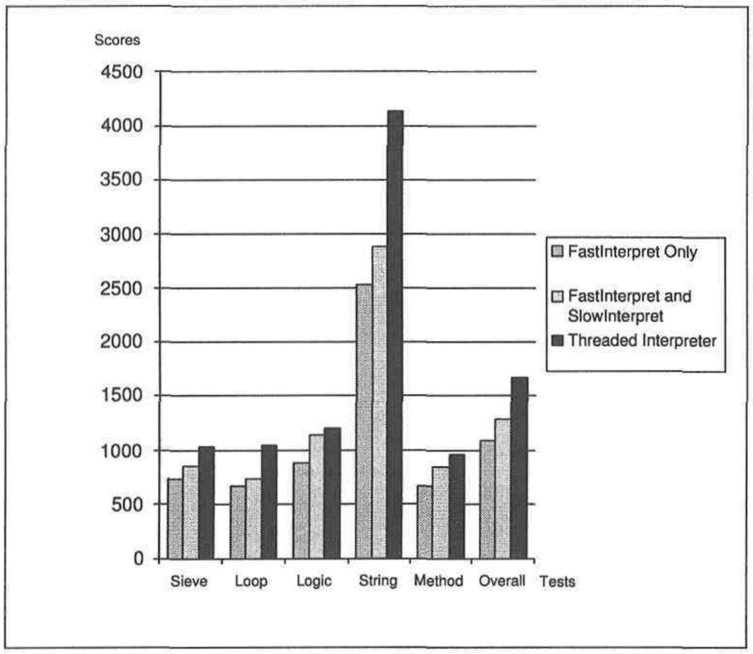


Figure 5.15: Threaded interpreter performance

perform a dynamic lookup for each subsequent invocation of the method *m* resulting in a significant overhead. The inline technique with the modified cache structure will simply test if the current receiver equals the one that is stored in the cache. Accordingly, the method definition will be retrieved from the cache for all subsequent invocations of the method *m* resulting in a significant speed-up.

5.3.4 Implementation & Results

In the sequel, we give the results related to the techniques we proposed. We have implemented and integrated a threaded interpreter in the KVM. The experimental results obtained using the CaffeineMark benchmark and highlighted in Figure 5.15, show an execution enhancement up to 53% with respect to the non-optimized main-loop (*FastInterpret*) and 30% with respect to the optimized main-loop (composed of two loops: *FastInterpret* with *SlowInterpret*) thanks to the threaded interpreter. Currently, other members of the Group are carrying out an in-progress work including the implementation of the interpreter codelets generation, the implementation of a compiler leveraging E-Bunny technology by reusing these codelets and a porting of the overall implementation to the ARM architecture.

As of the method call acceleration technique, for some typical examples (e.g. Java programs that frequently call inherited methods), it is capable to reach a speedup of

```
public class A {
  public void m() ;
}
public class B extends A {
  public void m() { };
}
public class C extends B { }
public class D extends C {
  public static void main(String args[]) {
    A o;
    o = new D();
    int i = 0;
    while (i < 1000000) {
      o.m();
      i++;
    }
  }
}
```

Figure 5.16: Typical example for our Optimization technique

more than 27%. Figure 5.16 shows a typical example that contains a class hierarchy and an invocation of an inherited method *m*. Figure 5.17 shows the execution time acceleration for this example. This time is given with respect to the hash table size. Figure 5.18 outlines the trade-off between footprint and collisions. In fact, when the hash table size increases, the footprint increases and the number of collisions decreases so the lookup is more accelerated. However, the footprint is critical in embedded systems, so we can not minimize so much the collisions. Our technique is flexible such that the footprint is tunable.

5.4 Conclusion

We reported, in this chapter, two techniques for the acceleration of embedded Java virtual machines. The first one relies on a tight collaboration between a threaded interpreter and a lightweight dynamic compiler that reuses some of the codelets of this interpreter. The second acceleration technique is related to the acceleration of the dynamic method lookup. The results show that our optimizations are efficient and not expensive from the footprint standpoint. Moreover, the proposed techniques are very generic and could be successfully applied to any embedded Java virtual machine.

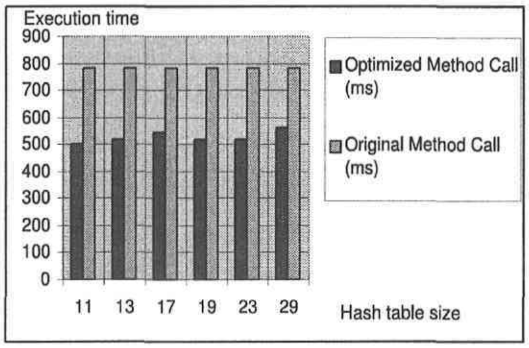


Figure 5.17: Execution time acceleration for a typical example

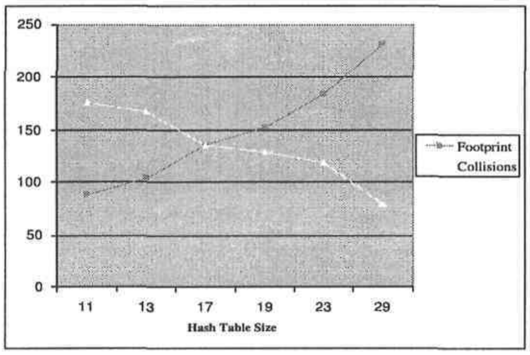


Figure 5.18: Trade-Off between footprint and collisions

Part II

Semantic Foundations of JVML/CLDC

Chapter 6

Concurrency Models

6.1 Introduction

Establishing the semantic correctness of an optimization technique consists of proving that the optimization preserves the semantics i.e. the original program and the optimized one are semantically equivalent. This entails the elaboration of one semantics if the original and optimized programs are both expressed in the same language. In the case of dynamic compilation, we are in the presence of two languages[1]: the source language and the target language. Hence, this means that two semantics are needed.

In the literature on programming languages, many researchers have used the method introduced by Morris in [90], further promoted in [130], to establish the correctness of a compilation/optimization process. This approach advocates the use of algebraic data types and algebraic semantics to capture the optimization correctness as the following equation:

$$encode(semantics_{source}(P_1)) = semantics_{target}(compile(P_1))$$

This amounts to the commutation of the diagram reported in Figure 6.1. Later, this approach was accommodated to use an operational semantic style as what Stephenson proposed in [122] or a denotational semantics style as what Wand proposed in [133]. In a denotational semantics setting, the correctness of the compiler is expressed as the equality of the denotation of the source program and the denotation of its translation:

$$semantics_{source}(P_1) = semantics_{target}(compile(P_1))$$

This paradigm for proving compiler correctness is outlined in Figure 6.2. Note that the presented figures outline the equivalence between a source program and a

[1]In our project, the source code is JVML/CLDC (Java Virtual Machine Language for Connected Limited Device Configuration) and the target code is the binary language of Intel processors.

$$\begin{tikzcd}
P_1 \arrow[r, "compile"] \arrow[d, "semantics_{source}"'] & P_2 \arrow[d, "semantics_{target}"] \\
Semantics_1 \arrow[r, "encode"] & Semantics_2
\end{tikzcd}$$

Figure 6.1: Morris Approach to Compiler Correctness

$$\begin{tikzcd}
P_1 \arrow[r, "compile"] \arrow[dr, "semantics_{source}"'] & P_2 \arrow[d, "semantics_{target}"] \\
 & Intermediate\ Language
\end{tikzcd}$$

Figure 6.2: Wand Paradigm for Proving Compiler Correctness

compiled one. However, they are valuable for proving the correctness of other kind of optimizations, which are also transformations of programs.

Hence, to establish the correctness of optimizations of JVML/CLDC programs, we have to provide a semantic model for JVML/CLDC, which is a concurrent language. Therefore, a concurrency model is needed to ascribe a semantics for this language. Concurrency models can be classified with respect to the following criteria [137]:

- System versus behavior: system models represent explicitly state information while behavioral models are more abstract and put focus only on the interactions in the system. The models that put focus on the system are called *intensional* while behavioral models are known as *extensional*.

- Branching versus non-branching: branching models take into consideration choice points, expressing non-determinism, that emerge at execution while non-branching models ignore them. There are two forms of non-determinism: bounded and unbounded. Bounded nondeterminism refers to the case where every terminating computation has only a finite number of possible options while in the unbounded non-determinism form, the set of options can be infinite [111]. Note that the exact definition of the word option depends on the studied language i.e. for instance an option can be a result if the language allows value return or a choice if it is not the case.

- Interleaving versus non-interleaving: interleaving models reduce the parallel composition of processes to a choice between possible interleaving executions of them.

On the other hand, non-interleaving (called also true concurrency models) offer the possibility for a simultaneous execution between the composed actions.

Concurrency models can also be classified with respect to the two most adopted semantic styles: operational and denotational. Before presenting operational and denotational semantic models, we provide some mathematical definitions that clarify the presentation.

6.2 Mathematical Definitions

This section is devoted to the presentation of some definitions that are useful to get a clear and full understanding about the content of the current and next chapters.

6.2.1 Some Notions about Domain Theory

A poset $(P,\sqsubseteq)$ is a non-empty set P equipped with a partial order relation $\sqsubseteq$. An element u of P is an upper bound of a subset S of P, if $s \sqsubseteq u$ for each $s \in S$. The dual definition of a lower bound of S is obtained using $\sqsupseteq$ instead of $\sqsubseteq$. An element u is the least upper bound (called also supremum) of a subset S of P if it is an upper bound of S and if $u \sqsubseteq t$ for each $t \in P$ upper bound of S. The definition of the greatest lower bound (infimum) is defined dually. A set D is directed if each finite subset $F \subseteq D$ has an upper bound in D. A set D is countably directed if every countable subset of D has an upper bound in D. A poset P is a directed complete partial order (dcpo) if each directed set D in P has a least upper bound. In this case, the least upper bound of D is denoted by $\sqcup D$.

A subset $F \subseteq P$ is filtered if each finite subset of F has a lower bound in F. A dcpo is a complete partial order (cpo) if it has a least element $\perp$. A subset S of a cpo P is bounded if there exists $x \in P$ such that $s \sqsubseteq x$ for each $s \in S$. A poset P is consistently complete if every non-empty subset X, for which each pair of elements has an upper bound in P, has a least upper bound.

Let P be a dcpo. An element $x \in P$ is compact if for each directed subset $D \subseteq P$, if $x \sqsubseteq \sqcup D$, $\exists k \in D$ such that $x \sqsubseteq k$. The set of compact elements of P is denoted by $K(P)$. We have for $x \in P$, $K(x) = K(P) \cap \downarrow x$, where $\downarrow x = \{y \in P \mid y \sqsubseteq x\}$. P is an algebraic domain if $K(x)$ is directed and $x = \sqcup K(x)$ for each $x \in P$. An element p of a dcpo P is prime if for each subset $X \subseteq P$, which has a least upper bound, $p \sqsubseteq \sqcup X$ implies $p \sqsubseteq x$ for some $x \in X$. The set of prime elements below some element $x \in P$ is denoted by $Pr(x)$. P is prime algebraic if $x = \sqcup Pr(x)$ for each $x \in P$.

A subset X of a cpo P is upward closed (called also upper set) if for each element of this set all greater elements are in the set i.e. we have $\uparrow X = X$ where $\uparrow X = \bigcup_{x \in X} \uparrow x$ and $\uparrow x = \{y \in P \mid x \sqsubseteq_P y\}$. A subset X of a cpo P is downward closed (called also lower set) if for each element of this set all smaller elements are in the set i.e $\downarrow X = X$ where $\downarrow X = \bigcup_{x \in X} \downarrow x$. A subset of a cpo P is convex if whenever a and c are in the set and $a \sqsubseteq_P b \sqsubseteq_P c$, b is in the set too. An ideal in a cpo P is a downward closed and directed subset in P. The ideal completion of a poset $(P,\sqsubseteq)$ is $(ideals(P),\subseteq)$ where $ideals(P)$ is the set of all the ideals in P.

A function $f : P \rightarrow Q$ between two posets P and Q is monotone if, whenever $x \sqsubseteq_P y$, then $f(x) \sqsubseteq_Q f(y)$.

A monotone function $f : P \rightarrow Q$ between posets P and Q is (Scott) continuous if, for every directed set D that has a least upper bound $\sqcup D \in P$, $\sqcup\{f(x) \mid x \in D\} = f(\sqcup D)$ i.e. f preserves least upper bounds of directed subsets. f is ω_1-continuous if it preserves least upper bounds of countably directed sets (ω_1 is the first uncountable ordinal).

$(S,+)$ is called a semilattice if S is a cpo and $+ : S \rightarrow S$ is a continuous operation.

A poset P is a local cpo (lcpo) if it has a least element $\perp$ and if every directed set of P that has an upper bound has a least upper bound.

Let $(P,\sqsubseteq)$ be a cpo, $f : P \rightarrow P$ a function. Then, $x \in P$ is a prefixed point of f if $f(x) \sqsubseteq x$. If $x \sqsubseteq f(x)$, then x is a postfixed point of f.

6.2.2 Some Notions about Category Theory

A category C is defined as a collection of objects together with a collection of morphisms (functions between objects) and which satisfies the following constraints:

- Composition is associative: Given $f : X \rightarrow Y$, $g : Y \rightarrow Z$ and $h : Z \rightarrow W$, $h \circ (g \circ f) = (h \circ g) \circ f$.
- For every object X there is an identity morphism $\mathsf{id}_X : X \rightarrow X$, satisfying $\mathsf{id}_X \circ g = g$ for every morphism $g : Y \rightarrow X$ and $f \circ \mathsf{id}_X = f$ for every morphism $f : X \rightarrow Y$.

Let S be a collection of objects $(X_i)_{i \in I}$ together with a collection of morphisms $(f_{ji} : X_j \rightarrow X_i)_{i \leq j}$ such that $\forall i \leq j \leq k.\ f_{ki} = f_{ji} \circ f_{kj}$. A cone over S is an object X together with a family of morphisms $(f_i : X \rightarrow X_i)_{i \in I}$ such that for $i \leq j$ we have $f_i = f_{ji} \circ f_j$.

6.2.3 Some Notions about Transfinite Numbers

Transfinite numbers, also known as infinite numbers, are numbers that are not finite. There are two different classes of transfinite numbers: ordinal and cardinal numbers. An ordinal number is a number, which is used to denote the position of an element in an ordered sequence, whereas a cardinal number denotes the size of a set. As an example of these numbers, we have the lowest transfinite ordinal number, which is ω (ω is the limit of the sequence 0, 1, 2, 3, 4, ...) and the first transfinite cardinal number, which is $\aleph_0$, aleph-null, (aleph-null is the cardinality of the infinite set of the integers).

There are two categories of ordinals: Limit ordinals and successor ordinals. Limit ordinals are ordinal numbers, which haven't direct predecessors. The other ordinals are called the successor ordinals. Ordinals can also be countable or uncountable. The smallest uncountable ordinal is equal to the set of all countable ordinals, and is usually denoted by ω_1.

6.3 Operational Models

Operational models aim at giving a formal description of a system through an abstract machine. In the literature, there are plenty of operational semantic models for concurrent languages. Famous examples of these semantics include operational semantics for CSP [103] and CCS [87]. We give hereafter, a brief description of Labelled Transition Systems, which are the most famous operational model.

A Labelled Transition System (LTS) is a tuple $\langle S, s_0, \mathcal{L}, T \rangle$ where:

- S is the set of the states,
- s_0 is the initial state,
- $\mathcal{L}$ is the set of labels and
- T is a transition relation between states such that $T \subseteq S \times \mathcal{L} \times S$.

LTS are used to provide a semantics for CCS [87]. In this semantic model, equivalence between processes is established via a bisimulation relation between them. This relation is interesting since it allows for example the comparison between two designs of a system. Another application of LTS is model checking of concurrent systems. In fact, various properties, such as safety and liveness, of concurrent systems can be checked in a logical framework. True concurrent LTS had emerged by the extension of LTS with an independence relation between states [13]. This extension allows to minimize state explosion.

The operational style is known to be too concrete since generally, low level details are considered in the elaboration of the semantics. This decreases the abstraction level of the semantic model. Accordingly, denotational models are meant to abstract from these details.

6.4 Denotational Models

Few denotational models for concurrency exist in the literature. In the sequel, we review these models.

6.4.1 Failure Sets

Failure sets [19] is a denotational model in which a process is denoted by a set of pairs called *failures*. The first component of a failure is an execution trace of the process in question while the second component is the set of events that are refused by it. More precisely, let Σ be a set of events, Σ^* the set of all possible traces, E an element of $\mathcal{P}(\Sigma^* \times \mathcal{P}(\Sigma))$, $b \in \Sigma$, $s,t \in \Sigma^*$ and $X,Y \in \mathcal{P}(\Sigma)$. Let $\langle\rangle$ be the empty trace and $\langle b\rangle$ be the trace containing b. E is a failure set if it satisfies the following conditions:

1. $(s,X) \in E \Rightarrow X$ finite,
2. $(\langle\rangle,\emptyset) \in E$,
3. $(st,\emptyset) \in E \Rightarrow (s,\emptyset) \in E$ (prefix-closed),
4. $(s,Y) \in E \wedge X \subseteq Y \Rightarrow (s,X) \in E$,
5. $(s,X) \in E \wedge (s\langle b\rangle,\emptyset) \notin E \Rightarrow (s,X \cup \{b\}) \in E$.

As an example, the process STOP, which is a process that never does anything, is denoted by the following failure set: $\{(\langle\rangle, X) \mid X \subseteq \Sigma \;\&\; X \text{ finite}\}$. In fact, this process cannot execute any event in Σ.

The failure sets model was first used to provide a denotational semantics for CSP. It was extended to deal with divergence[2]. In the failure/divergence model, a process is denoted by $\langle F,D\rangle$ where F is a failure set and D is the set of divergence traces (the traces after which a process diverges). Unfortunately, this model can not discriminate a process, which can communicate a finite number of a from the process that can communicate an infinite number of a. This discrimination issue is discussed in details

[2]A process is considered as divergent if it executes infinite internal invisible actions

in Section 6.5.1. Roscoe et al. [111] extended the failure/divergence model by adding infinite traces that a process can communicate. A process is thus denoted by $\langle F,D,I \rangle$. This model was devoted to the treatment of unbounded non-determinism.

6.4.2 Acceptance Trees

Acceptance trees [58] constitute the dual model of failure sets as proved in [15]. In this model, a process is denoted by a set of pairs called *acceptances*. The first component of an acceptance is an execution trace while the second component refers to the set of events that can be executed by the process. The events, which are outside this set are refused by the process. More accurately, let A be an element of $\mathcal{P}(\Sigma^* \times \mathcal{P}(\Sigma))$, $a \in \Sigma$, $\sigma,\mu \in \Sigma^*$ and $X,Y,Z \in \mathcal{P}(\Sigma)$. A is an acceptance set if it satisfies the following conditions:

1. $(\sigma,X) \in A \Rightarrow X \subseteq initials(A/\sigma)$,
2. $\sigma \in traces(A) \wedge a \in initials(A/\sigma) \Rightarrow \exists X \mid (\sigma,X) \in A \wedge a \in X$,
3. $((\sigma,X) \in A \wedge (\sigma,Y) \in A) \Rightarrow (\sigma,X \cup Y) \in A$ (union-closed),
4. $((\sigma,X) \in A \wedge (\sigma,Z) \in A \wedge X \subseteq Y \subseteq Z) \Rightarrow (\sigma,Y) \in A$ (convex-closed),
5. $\sigma\mu \in traces(A) \Rightarrow \sigma \in traces(A)$ (prefix-closed).

Where:

$$\begin{array}{lcl} traces(A) & = & \{\sigma \mid \exists X.\ (\sigma,X) \in A\} \\ initials(A) & = & \{a \mid \langle a \rangle \in traces(A)\} \\ A/\sigma & = & \{(\mu,X) \mid (\sigma\mu) \in traces(A)\} \end{array}$$

As an example, the acceptance tree $\{(a,\{b,c\}),(a,\{c\})\}$ denotes a process that already executed a and that accepts (i.e. can execute) either b or c. The process `STOP` is denoted by the following acceptance tree: $\{(\langle\rangle, \emptyset)\}$.

The acceptance trees model was first used to provide a denotational semantics for CSP. An extension of this model appeared in [15]. The aim of this extension is to capture imperative aspects of languages like CML. Another extension of acceptance trees for probabilistic processes can be found in [96].

Failure sets and acceptance trees give an interleaving meaning to parallelism. In fact, in these models, the parallel composition of two processes is defined as all the possible interleaving between them. This leads to a state explosion problem.

On the other hand, true concurrency models avoid this state explosion by considering that independent actions/events can be executed simultaneously. In the sequel, we review famous true concurrency models.

6.4.3 Event Structures

Event structures were introduced first in [135] to give an abstract representation of the behavior of petri nets. They constitute a model for concurrent computation that takes into account causal relations between events. More precisely, an event structure [135] is a tuple $\langle E,\preceq,\#,l,L\rangle$, where:

- E is a set of events partially ordered by $\preceq$, which is called a causality relation. $\preceq$ should satisfy the following constraint: $\forall e \in E.\ \{e' \mid e' \preceq e\}$ is finite.
- $\#$ is a symmetric and irreflexive relation called conflict relation. This relation specifies the events that could not occur in parallel. It should satisfy the following constraint: $\forall e,e',e'' \in E.\ e \# e' \wedge e' \preceq e'' \Rightarrow e \# e''$.
- L is the set of labels and l: $E \rightarrow L$ a labelling function.

As an example, for an event structure $\langle E,\preceq,\#,l,L\rangle$, where $V = \{e_1, e_2, e_3\}$, $\preceq = \{(e_1, e_1), (e_1, e_2), (e_2, e_2)\}$ and $\# = \{(e_1, e_3), (e_3, e_1), (e_2, e_3), (e_3, e_2)\}$, e_1 is executed before e_2 while e_1 and e_2 cannot be executed in parallel with e_3.

Some quantitative extensions of event structures such as [65] were proposed to capture real-time systems and reason about system performance. Particularly, famous extensions introduce the notions of time and probability in the event structure model.

It is worth to mention that event structures [135] and labelled transition systems with independence [13] can capture true concurrency but they lack explicit description of it. Lately, Gastin and Mislove [49, 50] provided an explicit description of true concurrency in a denotational resource-based model. In the sequel, we give a detailed description of two resource-based models. More details can be found in [49] and [50].

6.4.4 Resource Trace Model

The resource trace model was presented in [49]. The main motivation behind this work is to design a fully abstract semantic model, which is based on the resource concept. In what follows, we give an overview of this model.

Real Trace Definition

Let Σ be a finite set of actions, $\mathcal{R}$ a finite set of resources and $res : \Sigma \rightarrow \mathcal{P}(\mathcal{R})$. A dependence relation D is defined as follows:

$$D = \{(a,b) \in \Sigma \times \Sigma \mid res(a) \cap res(b) \neq \emptyset\}$$

$a\ D\ b$ means that a and b are dependent. Otherwise, they are independent and we write $a\ I\ b$ where I is the dual relation of D i.e. the set of independent action pairs.

A real trace set $\mathbb{R}(\Sigma,res)$ is the set of elements t such that t is a directed acyclic graph $[V,E,\lambda]$, where:

- V is a countable set of events (an event is an occurrence of an action),
- $E \subseteq V \times V$ is a synchronization relation on V,
- λ: $V \rightarrow \Sigma$ is a node-labelling function and
- All vertices have a finite set of predecessors. This means that each vertex should have a finite past i.e. $\forall p \in V.\ {\downarrow}p = \{q \in V \mid (q,p) \in E^*\}$ is finite. Note that E^* denotes the reflexive, transitive closure of E.

  If there exists a vertex having an infinite past, the trace will have a transfinite part and is not considered as a real trace.

The relation between D and E is defined as follows:

$$\forall p,q \in V.\ (\lambda(p),\lambda(q)) \in D \Leftrightarrow (p,q) \in E \cup E^{-1} \cup \{(p,p) \mid p \in V\}$$

Example We provide some examples that illustrate the coding of real traces:

- $[\{a_1,b_1\},\{(a_1,b_1)\},\{(a_1,a),(b_1,b)\}]$ is a real trace where:
  $\{a_1,b_1\}$ is the event set, $\{(a_1,b_1)\}$ is the synchronization relation, which states that a_1 precedes b_1 and $\{(a_1,a),(b_1,b)\}$ is the labelling function. Note that the authors use the set notation for the labelling function to clarify the presentation.
- $[\{a_1,b_1\},\emptyset,\{(a_1,a),(b_1,b)\}]$ is another example of a real trace. The events a_1 and b_1 are executed simultaneously since the synchronization set is empty, which means that there is no dependence between these two events.

Real Trace Concatenation

In what follows, we provide the formal definition of the concatenation of two real traces.

Let *alph* be the alphabet of a trace $t = [V,E,\lambda]$ and defined by $alph(t) = \bigcup_{v \in V} \lambda(v)$. The resources needed by a trace t are defined as follows:

$$res(t) \quad = \quad res(alph(t)) \quad = \quad \bigcup\{res(a) \mid a \in alph(t)\}$$

Let $resinf(t)$ be the resources at infinity of a trace t and $alphinf(t)$ the actions from Σ that occur infinitely in t. We have: $resinf(t) = res(alphinf(t))$. Let t_1 and t_2 be two real traces such that:

$$\begin{array}{lcl} t_1 & = & [V_1,E_1,\lambda_1], \\ t_2 & = & [V_2,E_2,\lambda_2] \text{ and} \\ resinf(t_1) & \cap & res(t_2) = \emptyset. \end{array}$$

The concatenation of t_1 and t_2 is defined as follows:

$$\begin{array}{lcl} t_1.t_2 & = & [V,E,\lambda] \text{ where:} \\ V & = & V_1 \uplus V_2, \\ \lambda & = & \lambda_1 \uplus \lambda_2 \text{ and} \\ E & = & E_1 \uplus E_2 \uplus (V_1 \times V_2 \cap \lambda^{-1}(D)) \end{array}$$

In the aforementioned definitions, $\uplus$ denotes the disjoint union and $\lambda^{-1}(D)$ denotes the set of pairs composed of events belonging to $V_1 \times V_2$ and which are dependent. More precisely, $\lambda^{-1}(D) = \{(e,e') \in V_1 \times V_2 \mid (\lambda(e),\lambda(e')) \in D\}$.

It is worth to mention that the condition $resinf(t_1) \cap res(t_2) = \emptyset$ allows to avoid getting transfinite traces in which an event has an infinite past set. More precisely, all the events of t_2 should be independent from any event, which is an occurrence of an action in $alphinf(t_1)$.

This condition is also considered as an important solution to the theoretical issue related to the monotonicity of the concatenation of traces. This idea was proposed first by Gastin and Teodosiu [52].

Prefix Order

To establish the recursion semantics, the authors defined an ordering between the elements of a domain. A prefix order $\preceq$ is defined over real traces as follows:

$$r \preceq t \quad \Leftrightarrow \quad \exists s \in \mathbb{R}(\Sigma,res) \text{ such that } t = r.s,\ s \text{ is unique and denoted by } r^{-1}t.$$

Informally, the trace t is doing more actions than r, which is considered as a prefix of t.

$$\left(a_1 \rightarrow b_1 \rightarrow c_1 \rightarrow e_1 \rightarrow e_2 \;\cdot\;\cdot\;\cdot\;, \qquad R \right)$$

Figure 6.3: Example of a resource trace

Resource Trace Definition

The resource trace set over $(\Sigma,\mathcal{R},res)$ is the family:

$$\mathbb{F}(\Sigma,res) \;=\; \{(r,\, R) \mid r \in \mathbb{R}(\Sigma,\, res),\, R \subseteq \mathcal{R} \wedge resinf(r) \subseteq R\}$$

The real part of $x = (r,R)$ is r. It represents a real trace and is denoted by $\mathsf{Re}(x)$. The imaginary part is R and it represents the needed resources for the continuation of the process. It is denoted by $\mathsf{Im}(x)$.

Example Let R be a resource set. We suppose that $res(e) \subseteq R$.

$$([\{a_1,b_1,c_1,e_1,e_2,\ldots\},\{(a_1,b_1),(b_1,c_1),(c_1,e_1),(e_1,e_2),(e_2,e_3),\ldots\},$$
$$\{(a_1,a),(b_1,b),(c_1,c),(e_1,e),(e_2,e),\ldots\}],R)$$

is a resource trace where the real part is the following real trace:

$$[\{a_1,b_1,c_1,e_1,e_2,\ldots\},\{(a_1,b_1),(b_1,c_1),(c_1,e_1),(e_1,e_2),(e_2,e_3),\ldots\},$$
$$\{(a_1,a),(b_1,b),(c_1,c),(e_1,e),(e_2,e),\ldots\}]$$

The imaginary part of this resource trace is R. Figure 6.3 outlines this resource trace.

An approximation order $\sqsubseteq$ is defined over $\mathbb{F}$ as follows:

$$(r,R) \sqsubseteq (s,S) \quad \Leftrightarrow \quad r \preceq s \wedge R \supseteq S \cup res(r^{-1}s)$$

This means that (s,S) is a process for which r is a prefix of s and all the resources it already used and it needs for the continuation are a subset of R.

Note that if the continuation resources set of a process is empty, the real part of this process is finite and the process is considered as terminated. Otherwise, the process is a non-terminated one.

Resource Trace Concatenation

A concatenation operation "." over resource traces is defined as follows:

$$(r,R).(s,S) \;=\; (r.\mu_R(s),\, R \cup S \cup \sigma_R(s))$$

$$\begin{tikzcd}
\Bigg(a_1 \arrow[r] & b_1 \arrow[r] & c_1 \arrow[r] & e_1 \arrow[r] & e_2 \cdots , res(e) \Bigg) . \Bigg(d_1 \arrow[r] & f_1 , \emptyset \Bigg) =
\end{tikzcd}$$

$$\begin{tikzcd}
\Bigg(a_1 \arrow[r] & b_1 \arrow[r] & c_1 \arrow[r] & e_1 \arrow[r] & e_2 \cdots , res(e) \cup res(f) \Bigg) \\
d_1 \arrow[ur] & & & &
\end{tikzcd}$$

Figure 6.4: Example of a concatenation of two resource traces

where $\sigma_R(s) = res((\mu_R(s))^{-1}s)$ and $\mu_R(s)$ is the maximal prefix u of s satisfying: $res(u) \cap R = \emptyset$

As mentioned previously, the intuition behind looking for the maximal prefix, which is not using some of the resources dedicated to the continuation of the second trace, is to guarantee that the concatenation is an internal operation of $\mathbb{R}$. This allows to avoid getting transfinite traces. Moreover, it ensures the monotonicity of the concatenation with respect to the order $\sqsubseteq$.

Example Let p and q be two resource traces such that:

- p is the following:
  $([\{a_1,b_1,c_1,e_1,e_2,\ldots\},\{(a_1,b_1),(b_1,c_1),(c_1,e_1),(e_1,e_2),(e_2,e_3),\ldots\},$
  $\{(a_1,a),(b_1,b),(c_1,c),(e_1,e),(e_2,e),\ldots\}],\{\rho\})$
- q is the following:
  $([\{d_1,f_1\},\{(d_1,f_1)\},\{(d_1,d),(f_1,f)\}],\emptyset)$
- We assume also that $res(a)=\{\alpha\}$, $res(b)=\{\beta,\gamma\}$, $res(c)=\{\gamma,\rho\}$, $res(e)=\{\rho\}$, $res(d)=\{\beta,\zeta\}$, $res(f)=\{\zeta,\rho\}$.

The maximal prefix of q that does not use ρ contains uniquely of the event d_1. Figure 6.4 outlines the concatenation of these two resource traces. The resource trace, which is the result of the concatenation of p and q, is:

$([\{a_1,b_1,c_1,d_1,e_1,e_2,\ldots\},\{(a_1,b_1),(b_1,d_1),(b_1,c_1),(c_1,e_1),(e_1,e_2),(e_2,e_3),\ldots\},$
$\{(d_1,d),(a_1,a),(b_1,b),(c_1,c),(e_1,e),(e_2,e),\ldots\}],\{\rho,\zeta\})$

Hereafter, we give some algebraic properties of $\mathbb{F}$, which are relevant to give a meaning to recursion in this model:

- $(\mathbb{F},\sqsubseteq)$ is a prime algebraic domain and $(1,\mathcal{R})$ is the least element of this domain. Note that 1 denotes the real trace $[\emptyset,\emptyset,\emptyset]$.

- The compact elements in $\mathbb{F}$ are the finite resource traces.

Gastin et Mislove [49] illustrate the concept of resource traces by ascribing a denotational semantics to a simple language. In the sequel, we give the syntax as well as the denotational semantics of this language.

Language Syntax

The syntax of the studied language $\mathcal{L}$ is the following:

$$p \ ::= \ \texttt{STOP} \ \mid \ a \ \mid \ x \ \mid \ p \circ q \ \mid \ p \parallel_C q \ \mid \ p \mid_R \ \mid \ rec\ x.p$$

The meaning of each syntactic construction is the following:

- `STOP`: the process that cannot execute any action and is claiming all the resources. It denotes a deadlock.
- a: a process that can execute an action a and which terminates normally.
- x: a process variable.
- $p \circ q$: a weak sequential composition (concatenation) of two processes.
- $p \parallel_C q$: a parallel composition of two processes that synchronize on a channel C.
- $p \mid_R$: a process that is restricted to execute actions needing just resources in R.
- $rec\ x.p$: a definition of a recursive process.

Denotational Semantics

The semantic interpretation function $[\![\ ]\!]$ is defined from $\mathcal{L}$ to $[\mathbb{F}^\Lambda \to \mathbb{F}]$ where $\mathbb{F}^\Lambda$ denotes a product on Λ-copies of $\mathbb{F}$ i.e. the domain of mappings from Λ to $\mathbb{F}$. Let ω be an n-ary operator of the language and $\tilde{\omega}$ be a map: $[\mathbb{F}^\Lambda \to \mathbb{F}]^n \to [\mathbb{F}^\Lambda \to \mathbb{F}]$.

The semantics of a compound process is defined as follows:

$$[\![\omega(p_1, \ldots, p_n)]\!] \quad = \quad \tilde{\omega}([\![p_1]\!], \ldots, [\![p_n]\!])$$

The authors proved in [49] that for each operator ω, $\tilde{\omega}$ is continuous. This proves that the semantic function $[\![\ ]\!]$ is continuous too. Note that this continuity is needed for the elaboration of the recursion semantics.

To lighten the notation, the authors use in what follows the same symbol for ω and $\tilde{\omega}$.

Constants and Variables

Let σ: $\Lambda \rightarrow \mathbb{F}$ a mapping from variables to complex pomsets. The denotational semantics of constants and variables are defined by the maps: $[\![\texttt{STOP}]\!]$, $[\![a]\!]$ and $[\![x]\!] \in [\mathbb{F}^\Lambda \rightarrow \mathbb{F}]$ and defined by:

$$\begin{array}{lcl} [\![\texttt{STOP}]\!](\sigma) & = & (1,\mathcal{R}) \\ [\![a]\!](\sigma) & = & (a,\emptyset) \\ [\![x]\!](\sigma) & = & \sigma(x) \end{array}$$

The semantics of STOP is a process that is not progressing and claiming all the resources specified in $\mathcal{R}$. The semantics of a simple action is a terminated process since the continuation resources set is $\emptyset$. Note that the trace containing just an event, which is an instance of the action a, is denoted by a.

Weak Sequential Composition

The weak sequential composition is defined as the concatenation of processes. The weak sequential composition of processes imposes an execution order on dependent actions while it allows independent actions to be executed in parallel.

More precisely, the semantics of the weak sequential composition of two processes is defined as follows:

$$\begin{array}{lcl} \circ & : & [\mathbb{F}^\Lambda \rightarrow \mathbb{F}]^2 \rightarrow [\mathbb{F}^\Lambda \rightarrow \mathbb{F}] \text{ where:} \\ & & (f_1 \circ f_2)(\sigma) = f_1(\sigma) \ . \ f_2(\sigma) \end{array}$$

Each f_i is a continuous map from $\mathbb{F}^\Lambda$ to $\mathbb{F}$. The resources of the resulted resource trace are defined by the following equation:

$$\forall (p,q) \in \mathbb{F}^2. \ res(p.q) = res(p) \cup res(q)$$

An example illustrating the semantics of the weak sequential composition is already provided in Figure 6.4 since, as specified above, the semantics of weak sequential composition is exactly the same as concatenation.

Restriction

A restriction for a process p to use a resource set R means that just the events using resources in R are considered while the events in p, which are not satisfying this condition are denied. The resources for the continuation of the restricted process are the intersection of the resources for the continuation of p and R.

More formally, let $\mathbb{F}_R = \{x \in \mathbb{F} \mid res(\mathsf{Re}(x)) \subseteq R\}$. $\mathbb{F}_R$ is the set of real traces that use only the resources in R. The restriction semantics is defined as follows.

$$\begin{array}{lcl} |_R & : & [\mathbb{F}^\Lambda \rightarrow \mathbb{F}] \rightarrow [\mathbb{F}^\Lambda \rightarrow \mathbb{F}] \text{ defined by} \\ (t|_R)\sigma & = & t(\sigma) \restriction_R \ \text{ where:} \end{array}$$

$$\Big(a \longrightarrow b \longrightarrow c_1 \longrightarrow c_2 \quad \cdot \quad \cdot \quad \cdot \quad , \quad res(c) \Big)|_{\{\mu\}} = \Big(a \longrightarrow b \quad , \{\mu\} \Big)$$

Figure 6.5: A restriction of a process execution to a set of resources

$$\begin{aligned} \restriction_R \quad &: \quad \mathbb{F} \rightarrow \mathbb{F} \\ \restriction_R \quad &= \quad g \circ f \quad \text{where:} \end{aligned}$$

$$\begin{aligned} f \quad &: \quad \mathbb{F} \rightarrow \mathbb{F}_R \text{ defined by } x \mapsto \sqcup\{y \in \mathbb{F}_R \mid y \sqsubseteq x\} \\ g \quad &: \quad \mathbb{F}_R \rightarrow \uparrow(1,\, R) \subseteq \mathbb{F} \text{ defined by } (s,S) \mapsto (s,S \cap R) \end{aligned}$$

Note that f is a mapping that associates to a real trace x the least upper bound of the set of elements smaller than x and that use just resources in R. g is a mapping that restricts the continuation resources of a resource trace to a subset of R.

Moreover, we have:

$$res(x|_R) \quad = \quad res(x) \cap R$$

Example Figure 6.5 outlines the restriction of a process to a set of resources $\{\mu\}$. We suppose that: $res(c) = \{\beta,\mu\}$, $res(a) = \{\mu\}$ and $res(b) = \{\mu\}$. As shown in Figure 6.5, all the events not using just the resource μ are denied, while the continuation resources are restricted to the set $\{\mu\}$.

Parallel Composition

The parallel composition with synchronization on a channel C (defined as a set of actions) consists of computing the events in each of the composed processes that are not using the resources of the set C and which are independent with the other process. These events are specified below in a set called Σ'_C. More precisely, the composed process trace is specified as the least upper bound of the trace pairs set, which is built using Σ'_C. This least upper bound is specified below in φ_C.

More accurately, let $x_1 = (s_1,S_1)$, $x_2 = (s_2,S_2)$ two resource trace, C a channel and $Sync_C(x_1,x_2)$ be a synchronization event set and defined as the set of pairs $(a_1,a_2) \in alph(s_1) \times alph(s_2)$ satisfying:

$$res(a_1) \cap C = res(a_2) \cap C = res(a_1) \cap res(x_2) = res(a_2) \cap res(x_1) \neq \emptyset$$

The semantics of the parallel composition of x_1 and x_2 with synchronization over the channel C is specified as follows:

Let

$$
\begin{array}{lcl}
\Sigma' & = & (\Sigma \cup \{1\})^2 \backslash \{(1,1)\} \\
res' & : & \Sigma' \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by } res'(a_1 \parallel a_2) = res(a_1) \cup res(a_2) \\
\Pi & : & \Sigma' \rightarrow \Sigma \text{ defined by } \Pi(a_1,a_2) = a_1 \parallel a_2 \\
\Sigma'_C(x_1,x_2) & = & \{(a_1,1) \in alph(s_1) \times \{1\} \mid res(a_1) \cap (C \cup res(x_2)) = \emptyset\} \cup \\
 & & \{(1,a_2) \in alph(s_2) \times \{1\} \mid res(a_2) \cap (C \cup res(x_1)) = \emptyset\} \cup \\
 & & Sync_C(x_1,x_2) \\
R_C(x_1,x_2) & = & \{r \in \mathbb{R}(\Sigma'_C(x_1,x_2),res') \mid \pi_i(r) \preceq \mathsf{Re}(x_i) \text{ for } i = 1,2\} \\
X_C(x_1,x_2) & = & \{(t,T) \in \mathbb{F}(\Sigma',res') \mid alph(t) \subseteq \Sigma'_C(x_1,x_2) \wedge \\
 & & \pi_i(t,T) \sqsubseteq x_i \text{ for } i = 1,2\} \\
\varphi_C(x_1,x_2) & = & \sqcup X_C(x_1,x_2) \\
 & = & (\sqcup R_C(x_1,x_2), S_1 \cup S_2 \cup res({r_1}^{-1}s_1) \cup res({r_2}^{-1}s_2)) \text{ where} \\
 & & r = \sqcup R_C(x_1,x_2) \wedge r_i = \pi_i(r)
\end{array}
$$

The parallel composition of two processes x_1 and x_2 is defined as follows:

$$x_1 \parallel_C x_2 \quad = \quad \Pi(\varphi_C(x_1,x_2))$$

Note that the alphabet set Σ' is composed of pairs so the definition of the functions *res* and *alph* are extended to sets of pairs. The definition of the function Π is extended to resource traces. The projection functions π_i extract the element at position i of an n-uplet.

Moreover, we the resources needed by the composed process are defined as follows:

$$res(x_1 \parallel_C x_2) \quad = \quad res(x_1) \cup res(x_2)$$

Example Figure 6.6 outlines the parallel composition of two processes with the condition: the actions a,b and c are pairwise independent. The synchronization is over the channel $C = \{c\}$. Since a and b are independent, the events which are occurrences of a are executed in parallel with the events which are instances of b. Moreover, the composed processes synchronize on the events c_i which are occurrences of the action c. The continuation resources of the result process are the union of the continuation resources of the composed processes.

Let

$$\begin{array}{lcl}
\Sigma' & = & (\Sigma \cup \{1\})^2 \backslash \{(1,1)\} \\
res' & : & \Sigma' \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by } res'(a_1 \parallel a_2) = res(a_1) \cup res(a_2) \\
\Pi & : & \Sigma' \rightarrow \Sigma \text{ defined by } \Pi(a_1,a_2) = a_1 \parallel a_2 \\
\Sigma'_C(x_1,x_2) & = & \{(a_1,1) \in alph(s_1) \times \{1\} \mid res(a_1) \cap (C \cup res(x_2)) = \emptyset\} \cup \\
 & & \{(1,a_2) \in alph(s_2) \times \{1\} \mid res(a_2) \cap (C \cup res(x_1)) = \emptyset\} \cup \\
 & & Sync_C(x_1,x_2) \\
R_C(x_1,x_2) & = & \{r \in \mathbb{R}(\Sigma'_C(x_1,x_2),res') \mid \pi_i(r) \preceq \mathsf{Re}(x_i) \text{ for } i = 1,2\} \\
X_C(x_1,x_2) & = & \{(t,T) \in \mathbb{F}(\Sigma',res') \mid alph(t) \subseteq \Sigma'_C(x_1,x_2) \wedge \\
 & & \pi_i(t,T) \sqsubseteq x_i \text{ for } i = 1,2\} \\
\varphi_C(x_1,x_2) & = & \sqcup X_C(x_1,x_2) \\
 & = & (\sqcup R_C(x_1,x_2), S_1 \cup S_2 \cup res({r_1}^{-1}s_1) \cup res({r_2}^{-1}s_2)) \text{ where} \\
 & & r = \sqcup R_C(x_1,x_2) \wedge r_i = \pi_i(r)
\end{array}$$

The parallel composition of two processes x_1 and x_2 is defined as follows:

$$x_1 \parallel_C x_2 \quad = \quad \Pi(\varphi_C(x_1,x_2))$$

Note that the alphabet set Σ' is composed of pairs so the definition of the functions res and $alph$ are extended to sets of pairs. The definition of the function Π is extended to resource traces. The projection functions π_i extract the element at position i of an n-uplet.

Moreover, we the resources needed by the composed process are defined as follows:

$$res(x_1 \parallel_C x_2) \quad = \quad res(x_1) \cup res(x_2)$$

Example Figure 6.6 outlines the parallel composition of two processes with the condition: the actions a,b and c are pairwise independent. The synchronization is over the channel $C = \{c\}$. Since a and b are independent, the events which are occurrences of a are executed in parallel with the events which are instances of b. Moreover, the composed processes synchronize on the events c_i which are occurrences of the action c. The continuation resources of the result process are the union of the continuation resources of the composed processes.

$$\left(a_1 \to c_1 \to a_2 \to c_2 \cdot \cdot \cdot \cdot \, , \quad res(a) \cup res(c) \right) \quad \|_{\{c\}}$$

$$\left(c_1 \to b_1 \to c_2 \to b_2 \cdot \cdot \cdot \cdot \, , \quad res(b) \cup res(c) \right) \quad =$$

$$\left(\begin{tikzcd} & & a_2 \arrow[rd] & & \\ a_1 \arrow[r] & c_1 \arrow[ru] \arrow[rd] & & c_2 & \cdot \cdot \cdot \cdot \, , \; res(a) \cup res(b) \cup res(c) \\ & & b_1 \arrow[ru] & & \end{tikzcd} \right)$$

Figure 6.6: The parallel composition of two processes

Note that in this model the parallel composition of two actions sharing some resources leads to a deadlock.

Recursion

The following theorem, proved by **Tarski**, **Knaster** and **Scott**, establishes the conditions for a recursive function to have a fixed point.

Theorem 6.4.1 (Tarski-Knaster-Scott) *If $f \colon P \to P$ is a continuous selfmap of a cpo P, then f has a least fixed point given by* $\mathsf{fix}(f) = \sqcup_{n \geq 0} f^n(\perp)$

We give, in the sequel, the recursion semantics.

$$[\![rec\ x.p]\!](\sigma) \quad = \quad \bigsqcup\nolimits_{n \geq 0} x_n \text{ where:}$$

$$x_0 = (1,R) \text{ with } R = res([\![p]\!](\sigma[x \mapsto (1,\emptyset)])),$$
$$\text{and } x_{n+1} = [\![p]\!](\sigma[x \mapsto x_n])$$

Since all the semantic functions are proved to be continuous and $\mathbb{F}$ is a cpo, the fixed point exists as states the theorem 6.4.1. However, the semantics of recursion does not use the least element of the domain $\mathbb{F}$ (which is $(1,\mathcal{R})$) since the use of this element gives more resources than what is actually claimed by the recursive process. In fact, $\mathcal{R}$ will be the imaginary part of the semantics of this process. Instead, the authors used another starting point for the recursion, which claims just the needed resources i.e. $res([\![p]\!](\sigma[x \mapsto (1,\emptyset)]))$. The following example provides an illustration of the recursion semantics.

Example The following example outlines a process that is executing indefinitely the action a.

Let $\Sigma = \{a,b,c\}$ with $res(a) = \{\alpha\}$, $res(b) = \{\alpha,\gamma\}$ and $res(c) = \{\gamma\}$. Consider the process $q = rec\ x.p$ with $p = a \circ x$. The semantics of p is the continuous map:

$[\![p]\!]$: $\mathbb{F}^\Lambda \to \mathbb{F}$ defined by $[\![p]\!](\sigma) = (a,\emptyset).\sigma(x)$

The semantics of q is a fixed point of the continuous selfmap:

ϕ: $\mathbb{F} \to \mathbb{F}$ defined by $\phi(x) = (a,\emptyset).x$

The fixed point is computed as follows:

$$\begin{aligned} x_0 &= (1, res([\![p]\!](\sigma[x \mapsto (1,\emptyset)]))) = (1,\ res((a,\emptyset))) = (1,\{\alpha\}) \\ x_{n+1} &= (a,\emptyset)\ .\ x_n \\ &= (a^{n+1},\{\alpha\}) \end{aligned}$$

This sequence is increasing and its least upper bound $x^\omega = (a^\omega,\{\alpha\})$ is the semantics of the process q (recall that ω is the first infinite ordinal). The obtained process claims the resource set $\{\alpha\}$ for its continuation. This complies with the condition imposing that the resources needed by the actions that occur infinitely often are included in the continuation resource set of the process.

Hereafter, we summarize the denotational semantics of the language $\mathcal{L}$ in the following rules:

$$\begin{aligned} [\![\mathtt{STOP}]\!](\sigma) &= (1,\mathcal{R}) \\ [\![a]\!](\sigma) &= (a,\emptyset) \\ [\![x]\!](\sigma) &= \sigma(x) \\ [\![p \circ q]\!](\sigma) &= [\![p]\!](\sigma)\ .\ [\![q]\!](\sigma) \\ [\![p \parallel_C q]\!](\sigma) &= [\![p]\!](\sigma) \parallel_C [\![q]\!](\sigma) \\ [\![p|_R]\!](\sigma) &= ([\![p]\!](\sigma))|_R \\ [\![rec\ x.p]\!](\sigma) &= (rec\ x.[\![p]\!])(\sigma) = \textstyle\bigsqcup_{n\geq 0} x_n \quad \text{where} \\ & \quad x_0 = (1,R) \text{ with } R = res([\![p]\!](\sigma[x \mapsto (1,\emptyset)])) \text{ and} \\ & \quad x_{n+1} = [\![p]\!](\sigma[x \mapsto x_n]) \end{aligned}$$

6.4.5 Resource Pomsets Model

The resource pomsets model[3] is an extension of the resource trace model. The main intent of this extension is to deal with strict sequential composition and hiding. In fact, as shown in the several examples that we provide later, strict sequential composition requires that each event of the left-hand process precedes each event of the right-hand

[3]A pomset is a shorthand of a partially ordered multiset

one. Moreover, in this model hiding an event from a trace means removing it from that trace. If the hidden event executes between two independent events, a dependence, which is not based on resources, between these independent events is created. Hence, these operators introduce dependencies between events without necessarily sharing some resources. This requires introducing some changes on the resource trace model, which consist mainly in weakening the equivalence between the relations D and E to an implication (see Section 6.4.4).

Real and Complex Pomsets

Let $\mathcal{R}$ be a non-empty countable infinite set of resources, Λ a set of variables and Σ defined as follows:

$$\begin{array}{lcl} \Sigma & = & Act \cup R_\Lambda \\ Act & = & \{a \in \mathcal{P}_f(\mathcal{R}) \mid a \neq \emptyset\} \\ R_\Lambda & = & \{\rho_x \mid x \in \Lambda\} \end{array}$$

Note that the alphabet set is composed of resource sets. The reason underlying this choice is to deal with resource hiding as explained later. Moreover, this set contains resource variables ρ_x. A resource variable represents an action of unwinding of recursion and is meant for observing divergence.

A real pomset is defined as a labelled partial order $r = (V,\preceq,\lambda)$ where:

- V is a countable set of vertices,
- $\preceq$: a partial order on V satisfying $\downarrow p$ is finite for each $p \in V$,
- λ: $V \rightarrow \Sigma$ assigns to each element of V an element of Σ. We also have:
  $\lambda(p) \cap \lambda(q) \neq \emptyset \Rightarrow p \preceq q$ or $q \preceq p$

The synchronization relation between events is relaxed in the sense that equivalence is transformed to an implication. This is due mainly, as mentioned previously, to the fact that strict sequential composition and hiding can introduce some dependencies between events that are not claiming the same resource.

The real pomset domain is denoted by $\mathbb{R}$. The mapping $res : \mathbb{R} \rightarrow \mathcal{P}(\mathcal{R})$ is defined by $res((V,\preceq,\lambda)) = \bigcup_{v \in V} \lambda(v)$. Moreover, the length of a real pomset $r = (V,\preceq,\lambda)$ is defined by $|r| = |V|$.

A complex pomset is defined as a pair (r,R) where:

- r is a real pomset,

- R is the set of resources needed for the continuation of r and
- $resinf(r) \subseteq R$ and $res(r) \cup R$ is finite.

The complex pomset domain is denoted by $\mathbb{C}$.

Prefix Order

Let $t_1 = (V_1,\preceq_1,\lambda_1)$ and $t_2 = (V_2,\preceq_2,\lambda_2)$ two real pomsets. The prefix relation is defined as follows:

$$\begin{aligned} t_1 \preceq t_2 \quad \Leftrightarrow \quad & V_1 \subseteq V_2,\ V_1 = \downarrow_{\preceq_2} V_1 \wedge \preceq_1 = \preceq_2|_{V_1 \times V_1} \quad \text{where} \\ & \downarrow_{\preceq_2} V_1 = \{p \in V_2 \mid \exists q \in V_1.\ p \preceq_2 q\} \end{aligned}$$

The meaning of the prefix order is similar to the one proposed in the resource trace model. This order is also used to establish the recursion semantics.

Denotational Semantics

The language studied by Gastin and Mislove in [50] has the following syntax:

$$p \ ::= \ \texttt{SKIP} \ \mid \ a \ \mid \ x \ \mid \ p\,;q \ \mid \ p \circ q \ \mid \ p \parallel_C q \ \mid \ p\backslash R \ \mid \ rec\, x.p$$

Note the presence of sequential composition (operator ;) and hiding (operator \) in this language. The informal meaning of other operators (weak sequential and parallel composition) is the same as what is presented previously in the resource trace model.

It is worth to mention again that to lighten the notation, the authors use in what follows the same symbol for each syntactic operator and its corresponding semantic operator.

Strict Sequential Composition

The strict sequential composition of two processes means that the left-hand process is executed before the right-hand one. If the left-hand one is a non-terminated process i.e. its continuation resources set is not empty, the right-hand one is not executed. In this case, the resulted process needs the resources of the right-hand for its continuation.

Let $s_1 = (V_1,\preceq_1,\lambda_1)$ and $s_2 = (V_2,\preceq_2,\lambda_2)$ two real pomsets. The strict sequential composition of s_1 and s_2 is:

$$s_1\,;s_2 \quad = \quad (V_1 \uplus V_2, \preceq_1 \uplus \preceq_2 \uplus V_1 \times V_2, \lambda_1 \uplus \lambda_2)$$

More precisely, let $x = (r,R)$ and $y = (s,S)$. The strict sequential composition of two complex pomsets x and y is defined as follows:

$$\left(\begin{tikzcd}[row sep=small] & a_1 \\ b_1 \arrow[ur] \arrow[dr] & \\ & c_1 \end{tikzcd} , \quad \emptyset \right) \quad ; \quad \left(\begin{tikzcd}[row sep=small] a_1 \arrow[dr] & \\ & b_1 \\ c_1 \arrow[ur] & \end{tikzcd} , \quad R \right) \quad =$$

$$\left(\begin{tikzcd}[row sep=small] & a_1 \arrow[r] \arrow[ddr] & a_2 \arrow[dr] & \\ b_1 \arrow[ur] \arrow[dr] & & & b_2 \\ & c_1 \arrow[r] \arrow[uur] & c_2 \arrow[ur] & \end{tikzcd} , \quad R \right)$$

Figure 6.7: Example about strict sequential composition

$$x;y \quad = \quad \begin{cases} (r;s,S) & \text{if } R=\emptyset; \\ (r, R \cup res(y)) & \text{otherwise.} \end{cases}$$

Example Figure 6.7 outlines an example showing the strict sequential composition of two processes. For the sake of clarity, we don't show all the dependencies between these processes.

Weak Sequential Composition

In what follows, we provide the semantics of weak sequential composition, which has the same informal meaning as that presented in the resource trace model. Accordingly, we omit providing examples. The reader can see Figure 6.4 that outlines the semantics of weak sequential composition.

Let $s_1 = (V_1,\preceq_1,\lambda_1)$ and $s_2 = (V_2,\preceq_2,\lambda_2)$ be two real pomsets. If $resinf(s_1) \cap res(s_2) = \emptyset$, then the weak sequential composition of s_1 and s_2 is $s_1 \circ s_2 = (V,\preceq,\lambda)$ where:

- $V = V_1 \uplus V_2$,
- $\preceq$ is the transitive closure of $\preceq_1 \uplus \preceq_2 \uplus \{(p,q) \in V_1 \times V_2 \mid \lambda(p) \cap \lambda(q) \neq \emptyset\}$ and
- $\lambda = \lambda_1 \uplus \lambda_2$

Let f be the following mapping:

$$\begin{array}{llll} f\colon & \mathcal{P}(\mathcal{R}) \times \mathbb{C} & \rightarrow & \mathbb{C} \\ & (R,y) & \mapsto & f_R(y) = \sqcup\{x \in \mathbb{C} \mid x \sqsubseteq y \wedge \ res(\mathsf{Re}(x)) \cap R = \emptyset\} \end{array}$$

The semantics of weak sequential composition is as follows:

$$x \circ y \quad = \quad x \circ f_{\mathsf{Im}(x)}(y) = (\mathsf{Re}(x) \circ \mu_{\mathsf{Im}(x)}(y), \mathsf{Im}(x) \cup \sigma_{\mathsf{Im}(x)}(y))$$

$$\mu_R(y) \quad = \quad \sqcup\{r \in \mathbb{R} \mid r \preceq \mathsf{Re}(y) \wedge res(r) \cap R = \emptyset\}$$

$$\sigma_R(y) \quad = \quad \mathsf{Im}(y) \cup res(\mu_R(y)^{-1}\,\mathsf{Re}(y))$$

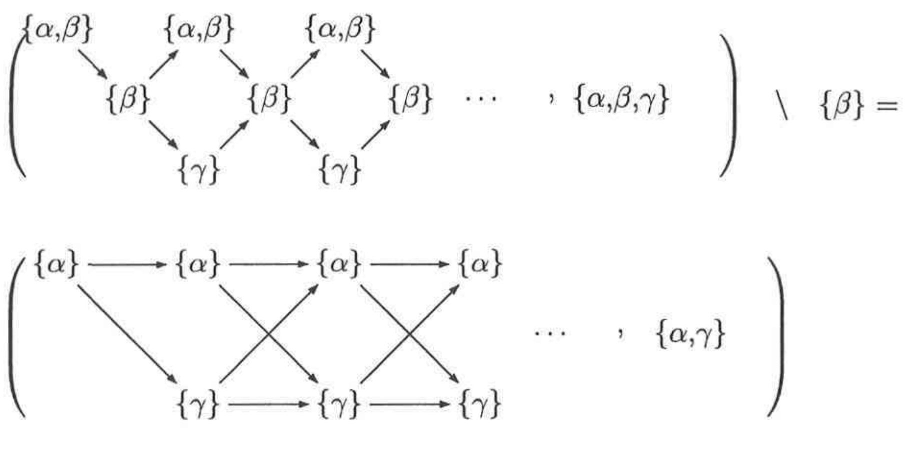


Figure 6.8: Example about hiding a process

Hiding

Hiding a set of resources R from a process p means removing from the labelling function all resources belonging to R and removing also those vertices that are associated with an empty resource label. The continuation resources of the result is composed of the continuation resources of p from which the set R is removed.

Let $s = (V,\preceq,\lambda) \in \mathbb{R}$. $s\backslash R = (V',\preceq',\lambda')$ where $V' = \{p \in V \mid \lambda(p)\backslash R \neq \emptyset\}$, $\preceq' = \preceq \cap (V' \times V')$ and $\lambda'(p) = \lambda(p)\backslash R \quad \forall p \in V'$.

Let $x = (s,S) \in \mathbb{C}$ be a complex pomset and $R \subseteq \mathcal{R}$. The hiding semantics is defined as follows:

$$x\backslash R \quad = \quad (s\backslash R, S\backslash R)$$

Example Figure 6.8 outlines an example about the hiding of a process. We see the use of resource sets (see the definition of Σ) in the trace and the creation of new dependencies after hiding the resource set $\{\beta\}$.

Parallel Composition

In the sequel, we provide the semantics of parallel composition, which has the same informal meaning as that presented in the resource trace model. Accordingly, we omit providing an example. The reader can see Figure 6.6 that outlines the semantics of parallel composition.

Let $s_1 = (V_1,\preceq_1,\lambda_1)$ and $s_2 = (V_2,\preceq_2,\lambda_2)$ be two real pomsets. The parallel composition of these real pomsets is defined as follows:

$$s_1 \parallel s_2 \quad = \quad (V_1 \cup V_2, (\preceq_1 \cup \preceq_2)^*, \lambda_1 \cup \lambda_2)$$

Let $\mathbb{R}_C(x_1,x_2)$ be a subset of $\mathbb{R}^2$ and C a subset of Σ such that each $(r_1,r_2) \in \mathbb{R}_C(x_1,x_2)$ satisfy the following conditions:

- $r_1 \preceq \mathsf{Re}(x_1)$ and $r_2 \preceq \mathsf{Re}(x_2)$,
- The number of occurrences of the action c is the same in r_1 and r_2 i.e. $|r_1|_c = |r_2|_c$ $\forall c \in C$,
- $\forall a \in alph(r_1)$ we have $a \in C$ or ($a\ I\ C$ and $a\ I\ x_2$) and $\forall a \in alph(r_2)$ we have $a \in C$ or ($a\ I\ C$ and $a\ I\ x_1$) and
- $r_1 \parallel r_2$ is a real pomset.

Let $x_1 = (s_1,S_1)$, $x_2 = (s_2,S_2) \in \mathbb{C}$ and $(r_1,r_2) = \sqcup\mathbb{R}_C(x_1,x_2)$. The parallel composition of x_1 and x_2 is the following:

$$x_1 \parallel_C x_2 \quad = \quad (r_1 \parallel r_2, S_1 \cup res(r_1^{-1}s_1) \cup S_2 \cup res(r_2^{-1}s_2))$$

Like in the resource trace model, the parallel composition of two actions sharing some resources leads to a deadlock.

Recursion

The complex pomset domain has no least element. In fact, $(1,\mathcal{R})$ is not the least element since $\mathcal{R}$ is infinite and so $(1,\mathcal{R})$ does not satisfy the third constraint of the definition of a complex pomset. Hence, $(1,\mathcal{R}) \notin \mathbb{C}$. The recursion semantics has some common points with that defined for the resource trace model. The introduction of ρ_x is the new element in this semantics. This resource is used to show unwinding i.e. the act of replacing a variable by a complex pomset.

More formally, let $\mathcal{D}$ be the domain of continuous maps from $\mathbb{C}^\Lambda$ to $\mathbb{C}$. We recall σ is a mapping from variables to processes (elements of $\mathbb{C}$). The recursion semantics is the following:

$$\begin{array}{lcl}
rec\ x & : & \mathcal{D} \to \mathcal{D} \\
(rec\ x.\ f)(\sigma) & = & \bigsqcup_{n \geq 0} x_n(f,\sigma) \quad \text{where} \\
 & & \\
x_0(f,\sigma) & = & (1, R_x(f,\sigma)) \\
 & & \\
R_x(f,\sigma) & = & res(\Phi_x(f,\sigma,(1,\emptyset))) \\
 & & \\
\Phi_x(f,\sigma,y) & = & \rho_x; f(\sigma[x \mapsto y]) \\
 & & \\
x_{n+1}(f,\sigma) & = & \Phi_x(f,\sigma,x_n(f,\sigma)) \\
 & = & \rho_x; f(\sigma[x \mapsto x_n(f,\sigma)])
\end{array}$$

We give an example illustrating the recursion semantics. The following process executes an action a indefinitely.

$$\begin{array}{lcl} \text{Let } p & = & a;x \\ R_x & = & res(\rho_x;a;x(\sigma[x \mapsto (1,\emptyset)])) \\ & = & \{\rho_x\} \cup res((a,\emptyset);(1,\emptyset)) \\ & = & \{\rho_x\} \cup res((a,\emptyset)) \\ & = & \{\rho_x,a\} \end{array}$$

The semantics of the process $q = rec\ x.\ p$ is the least upper bound of the chain $(x_i)_{i\in\mathbb{N}}$, which is the fixpoint of the recursion. In this case the fixpoint of the recursion is $x_w = ((\rho_x \rightarrow a)^w, \{\rho_x, a\})$, which means that the process q does unwinding then executes the action a indefinitely. Hereafter, we give the details about the computation of the fixpoint.

$$\begin{array}{lcl} x_0 & = & (1,\{\rho_x,a\}) \\ x_1 & = & \rho_x;[\![a;x]\!](\sigma[x{\mapsto}(1,res(a) \cup res(\rho_x))]) \\ & = & \rho_x;[\![a;x_0]\!] \\ & = & \rho_x;[\![a;(1,\{\rho_x,a\})]\!] \\ & = & (\rho_x \rightarrow a,\ \{\rho_x,a\}) \\ x_2 & = & (\rho_x \rightarrow a \rightarrow \rho_x \rightarrow a,\ \{\rho_x,a\}) \\ \ldots & & \\ x_w & = & ((\rho_x \rightarrow a)^w,\{\rho_x,a\}) \end{array}$$

6.5 Our Strategy

Many advantages cater for the adoption of a denotational semantic style for JVML/-CLDC. In fact, the denotational semantics has strong mathematical foundations. Moreover, it is compositional and more abstract than operational semantics. Furthermore, this style is very adequate for proving optimizations correctness. Actually, we can encode the semantics of the source and target languages in the same model. This is however not possible in an operational setting if the source and target languages are different. It is worth to mention that, in the related work, there is no denotational semantics for JVML or JVML/CLDC. Indeed, what we find in the literature is just a denotational semantics for a subset of the Java source language that excludes parallelism [5]. The main reason behind this lack is the difficult issues that emerge when one tries to design such a semantics. We intend to devise a concurrency model that can be accommodated for JVML/CLDC. More precisely, we found the resource pomsets model relevant for our research. In fact, the main interesting feature of the resource pomsets model is the concept of resources, which plays a fundamental role to know if two events can be executed simultaneously. This concept is useful since we aim to provide a semantics that describes in an explicit way the JVML/CLDC synchronization mechanism. For instance, the resources can be used to denote objects that can be locked by processes. Unfortunately, the resource pomsets model reduces parallel composition of events sharing some resources to a deadlock. This makes the model inadequate for JVML/CLDC and leads to the need for extending this model to include non-determinism. More details about the need for introducing non-determinism in the resource pomsets model are provided in the next chapter.

6.5.1 Non-Determinism

In what follows, we provide a brief overview about domain constructions and theoretical issues related to non-determinism.

Powerdomains

Non-determinism is behind the emergence of powerdomain constructions. A powerdomain of a domain D is a domain containing some of the subsets of D. There are three known powerdomains in the literature, which we define in what follows.

Let $\mathcal{P}_{<\omega}K(D) = \{F \mid \emptyset \neq F \subseteq K(D)\}$ be the finite non-empty subsets of $K(D)$, where $K(D)$ is the set of compact elements of D.

- *Hoare Powerdomain:*

  The Hoare (Lower) powerdomain is the ideal completion of $(\mathcal{P}_{<\omega}K(D),\sqsubseteq_L)$ where $F \sqsubseteq_L G \Leftrightarrow F \subseteq \downarrow G$. The semilattice operation is union and the continuous injection is $\eta_L(x){=}\downarrow x$. $\sqsubseteq_L$ is defined as follows:

  $X \sqsubseteq_L Y \quad \Leftrightarrow \quad \forall x \in X.\ \exists y \in Y.\ x \sqsubseteq y$

- *Smyth Powerdomain:*

  The Smyth (Upper) powerdomain is the ideal completion of $(\mathcal{P}_{<\omega}K(D),\sqsubseteq_U)$ where $F \sqsubseteq_U G \Leftrightarrow G \subseteq \uparrow F$. The semilattice operation is union and the continuous injection is $\eta_U(x){=}\uparrow x$. $\sqsubseteq_U$ is defined as follows:

  $X \sqsubseteq_U Y \quad \Leftrightarrow \quad \forall y \in Y.\ \exists x \in X.\ x \sqsubseteq y$

- *Plotkin Powerdomain:*

  The Plotkin (Convex) powerdomain is the ideal completion of $(\mathcal{P}_{<\omega}K(D),\sqsubseteq_C)$ where $F \sqsubseteq_C G \Leftrightarrow F \sqsubseteq_L G$ and $F \sqsubseteq_U G$. The semilattice operation is the convex hull of the union (i.e. $X \oplus Y = \downarrow(X \cup Y) \cap \uparrow(X \cup Y)$) and the continuous injection is $\eta_L(x){=}\{x\}$. $\sqsubseteq_C$ is defined as follows:

$$\begin{aligned} X \sqsubseteq_C Y &\Leftrightarrow \forall x \in X.\ \exists y \in Y.\ x \sqsubseteq y \text{ and } \forall y \in Y.\ \exists x \in X.\ x \sqsubseteq y \\ &\Leftrightarrow X \sqsubseteq_L Y \wedge X \sqsubseteq_U Y \end{aligned}$$

The three powerdomains are the classical constructions, which are used to deal with non-determinism. However, these constructions present a serious problem when it comes to deal with unbounded non-determinism. We discuss this issue in what follows.

Theoretical Issues related to Non-Determinism

As mentioned previously, there are two forms of non-determinism: bounded and unbounded. Bounded non-determinism limits the expressiveness of the studied language [111]. However, the semantic model remains simple and the proofs are relatively easy with respect to the unbounded form. On the other hand, unbounded non-determinism provides the capability to deal with more expressive languages. However, there are many theoretical issues that are related to the handling of unbounded non-determinism. In this context, Mislove [88] did a very interesting study about dealing with unbounded non-determinism in the aforementioned powerdomains. He proved that among the three powerdomains, the Smyth powerdomain is the only one for which we can find a representative domain i.e. a domain, which allows to have a semantics that discriminates between a process that do an infinite a from that performing any finite sequence of actions a and terminates normally. This is very important since the aim of supporting unbounded non-determinism is to have such a discrimination.

More precisely, none of these powerdomains allows to differentiate between the processes $\oplus_{n\in\mathbb{N}} a^n\checkmark$ and $\oplus_{n\in\mathbb{N}} a^n\checkmark + a^\omega$ [4]. It is worth to mention that an unbounded sum exists in the lower, upper and convex powerdomain and is respectively equal to the supremum, infimum and convex hull of the sets over which the sum is being formed. However, for instance for the lower powerdomain, the supremum of $\{\downarrow a^n\checkmark \mid n \in \mathbb{N}\}$ contains $\downarrow a^\omega$. The other powerdomains fail also to do such discrimination. More details about this issue can be found in [88]. The solution to this problem consists of leaving the realm of complete partial orders and continuous functions and using the realm of posets and monotone functions. This is a solution that is adopted in [111] and [89] for providing a denotational semantics for unbounded non-determinism to the languages CSP and timed CSP.

The following theorem provides an example of a construction of a representative model for an algebraic domain.

[4]$\checkmark$ is a symbol, which does not belong to the actions set and which is used to specify programs, which terminate normally

Theorem 6.5.1 (Mislove[88]) *If P is an algebraic domain, then* $P_{US}(P) = \{X \subseteq P \mid \emptyset \neq X \wedge X = \uparrow X\}$ *with* $X + Y = X \cup Y$ *and* $X \sqsubseteq Y$ *iff* $Y \subseteq X$ *is a representative local cpo for P.*

This construction will be the basis of our semantic model as it will be shown in the next chapter.

Unbounded non-determinism creates other challenges at the semantic level. In fact, in the presence of random assignment in the language under study, the semantic functions are not Scott-continuous [6]. Accordingly, a weaker notion of ω_1-continuity is considered. More precisely, more than ω steps are required to compute a fixed point for a recursive semantic function. Another issue that emerges in the presence of unbounded non-determinism is the invalidity of the standard inverse-limit reflexive domain construction technique [56]. An alternative transfinite reflexive domain construction technique was proposed in [53].

On the other hand, the treatment of unbounded non-determinism has many advantages. In fact, non-determinism allows to deal with very expressive languages. Another feature of unbounded non-determinism is that it allows to abstract from implementation details. This is one of the most reasons that cater for the adoption of this form of non-determinism when we elaborated a semantic model that can be accommodated to JVML/CLDC. Full details about the reasons underlying the choice of unbounded non-determinism and about our semantic model are provided in the next chapter.

Chapter 7

A Semantic Model for True-Concurrency with Unbounded Non-Determinism

7.1 Introduction

In the previous chapter, we studied several concurrency models and provided the reasons underlying the choice of the resource pomsets model as a starting point of our work. We pinpointed that the resource pomsets model reduces parallel composition of events sharing some resources to a deadlock. This makes the model inadequate for JVML/CLDC. In fact, non-determinism emerges in the execution of some JVML/CLDC programs. For instance, if two JVML/CLDC threads try to execute a synchronized block or method at the same time, the virtual machine interpreter allows just one of them to execute this block or method. The second is executed when the first leaves this block or method. The execution order of these threads depends on many parameters: time, processor speed, thread priority, etc.

An example of a non-deterministic embedded Java [1] program is the following:

```
import java.util.*;
public class Nondeterminism extends Thread {
public static Object l1 = new Object();
public static long i;

public static void main(String[] a) {
   Thread t1 = new Thread1();
   Thread t2 = new Thread2();
   i = System.currentTimeMillis();
```

[1]Embedded Java is the source language of JVML/CLDC

```
      if ((i%2)==0)
        i = 0;
      else
        i = 1;
      t1.start();
      t2.start();
   }
   private static class Thread1 extends Thread {
      public void run() {

          try { if (i>0)
                  Thread1.sleep(i); }
          catch (InterruptedException e) {}

         synchronized (l1) {
            System.out.println("Thread 1: Holding lock 1...");
         }
      }
   }
   private static class Thread2 extends Thread {
      public void run() {
         synchronized (l1) {
            System.out.println("Thread 2: Holding lock 1...");
            }
      }
   }
}
```

The semantics of this program is a choice between the possible sequential compositions of Thread1 and Thread2. Indeed, Thread1 can execute the synchronized block (starting with the instruction "synchronized (l1)" in the method run() of the class Thread1) before Thread2 or the opposite can happen. These two possible behaviors are illustrated by the following possible outputs of the program:

"Thread 1: Holding lock 1..."
"Thread 2: Holding lock 1..."

or

"Thread 2: Holding lock 1..."
"Thread 1: Holding lock 1..."

The interleaving of these two threads proves the presence of non-determinism at the semantic level for JVML/CLDC. This leads, as previously mentioned, to the need for extending the resource pomsets model by non-determinism. As mentioned in the previous chapter, there are two forms of non-determinism: bounded and unbounded and it is well-established that fairness is tightly connected to unbounded non-determinism

[6]. Fairness is a good means for abstracting away from implementation details when we reason about the semantics of concurrent languages. For instance, assuming that a process scheduler is fair, we can reason about the semantics of parallel composition of programs without the need for including implementation details such as thread priority, processor speed, etc. Let us consider the following embedded Java program, which illustrates a competition between two threads for the execution of a synchronized block:

```
import java.util.*;
public class Unbounded extends Thread {
public static Object l1 = new Object();
public static boolean b=true;
public static long i;
public static void compete() {
    Thread t1 = new Thread1();
    Thread t2 = new Thread2();
    i = System.currentTimeMillis();
    if ((i%2)==0)
      i = 0;
    else
      i = 1;
    t1.start();
    t2.start();
    t1.interrupt();
    t2.interrupt();
  }
public static void main(String[] a) {
 int x = 0;
 while(b)
 {
   compete();
   if (b)
     x = x+1;
  }
 System.out.println("Final Value of x");
 System.out.println(x);
}
private static class Thread1 extends Thread {
   public void run() {
       try { if (i>0)
               Thread1.sleep(i); }
       catch (InterruptedException e) {}
       synchronized (l1) {
         b=false;
      }
   }
}
private static class Thread2 extends Thread {
   public void run() {
```

```
            synchronized (l1) {
              b=true;
            }
        }
    }
}
```

The final value of x is known only if the boolean b is equal to `false`. This occurs when Thread2 executes the synchronized block before Thread1. By fairness hypothesis, i.e. assuming the scheduler is fair, this program is guaranteed to terminate, i.e. Thread2 is executed before Thread1. However, we have no knowledge about the exact final value of x. Actually, the set of possible values of x is infinite. This example shows that assuming fairness, we can reason easily about the semantics of embedded Java programs. Hence, by abstracting away from implementation details, we are able to get an idea about the possible behavior of an embedded Java program and consequently about a JVML/CLDC program. The example shows also the emergence of unbounded non-determinism since the set of possible results is infinite.

However, despite the abstraction power provided by unbounded non-determinism, there is no doubt that the semantic model becomes more complex than the one elaborated in the bounded case and mainly proofs become more difficult to do. In fact, the main issue in the treatment of unbounded non-determinism, is related to the Scott-continuity of the semantic operators [6]. The lack of Scott-continuity creates additional challenges to prove the existence of fixed points for recursive semantic functions.

The elaboration of a denotational semantic model for JVML/CLDC including unbounded non-determinism is, as illustrated above, a very challenging research. We give here, a full description of the extension of the resource pomsets model of Gastin and Mislove [50] by including unbounded non-determinism. Other researchers [66], in our group, are working on the accommodation of this model to JVML/CLDC. The starting point of their research is the current work.

In what follows, we present the language syntax, the process space and its algebraic properties, then the semantics with examples that illustrate our semantic functions. Moreover, we provide the formal proofs about the monotonicity of our semantic functions together with the proof about the existence of a fixed point for recursion. Finally, we give some concluding remarks.

7.2 Language Syntax

In this section, we present the language that we study and use to illustrate later our semantics. The syntax of this language is the following:

$$\mathcal{L} \ni P \quad ::= \quad \texttt{SKIP} \mid \texttt{STOP} \mid a \mid X \mid P\backslash R \mid P; P \mid \oplus_{i\in I} P_i \mid P \circ P \mid P \parallel P \mid rec\, X.\ P$$

Where:

- `SKIP` denotes a normal termination.
- `STOP` denotes a deadlock.
- The term a denotes a simple action belonging to Σ (a non-empty finite set of actions). There are some special invisible actions τ, which denote internal actions. We suppose that we have a finite set of invisible actions that we call Act_τ. More precisely, we assume that each visible action a has a dual invisible action τ that uses the same resource set as a. The set of visible actions is denoted by Act. Hence, we have $\Sigma = Act \cup Act_\tau$. We suppose that V is the set of all possible events and defined as follows: $V = \Sigma \times \mathbb{N}$. To lighten the notation, an event (a, i) is denoted by a_i.
- X denotes a process variable that belongs to a set of variables ζ.
- $P\backslash R$ denotes a hiding in P of events using a subset of the resource set R. Intuitively speaking, an event is hidden if the set of the resources it uses is a subset of R.
- $P; P$ denotes a strict sequential composition of two processes. All the events of the first process should occur before those of the second.
- $\oplus_{i\in I} P_i$: an unbounded countable non-deterministic choice between processes. We assume that the unbounded non-determinism is countable to avoid more complex theoretical issues.
- $P \circ P$ denotes a weak sequential composition between two processes. The events of the two processes are executed in parallel when there is no resource dependence, while dependent events are executed sequentially in favor of the left-hand process.
- $P \parallel P$ denotes a parallel composition between two processes. The events of the two processes are executed in parallel when there is no resource dependence. Dependent events are executed sequentially without giving a favor to a particular process.
- $rec\ X.\ P$ denotes a recursive definition of a process.

7.3 Process Space

In this section, our concern is to present the construction of the process space. This space is the set of denotations that will be associated to the syntactic terms of the language $\mathcal{L}$. In what follows, we define step by step our process space, which is the space of non-deterministic processes. First, we present the space of dependence maps, which is used to define the space of deterministic processes. The latter is used to define the space of non-deterministic processes.

7.3.1 Dependence Maps

The use of recursive spaces allows to have a high level of abstraction in the design of language semantics. Actually, the associated semantic functions are recursive, compact and help in getting clear proofs. This motivated us to design a recursive space, called $\mathbb{M}$, for dependence graphs. Besides, the recursiveness feature allows us to get a very simple and clear prefix relation ($\rhd$) between the elements of the space $\mathbb{M}$ as it is shown later. More precisely, an element of the space $\mathbb{M}$ is a map that associates, a pair (p_e,e), where p_e is a finite set of events representing the direct predecessors of the event e, with another map that represents the successors of e. The definition of our space requires the use of a transfinite space construction technique since the number of successors of an event can be transfinite. This is illustrated by the semantics of our concatenation operator $\circ$, which allows such transfinite dependence as it will be shown in Section 7.5.5.

More accurately, let $\rightarrow_{\omega_1}$ be the constructor of infinite maps in which an element can be associated with a transfinite number of elements. The space $\mathbb{M}$ is defined as follows:

$$\mathbb{M} \quad = \quad \mathcal{P}_f(V) \times V \rightarrow_{\omega_1} \mathbb{M}$$

The existence proof of $\mathbb{M}$ is based on the transfinite recursive space construction technique, proposed by Di Gianantonio et al. [53]. To simplify the presentation, the full details about the construction and the existence proof of $\mathbb{M}$ are provided in Section 7.6.

The space $\mathbb{M}$ is endowed with an ordering $\rhd$, which is defined as follows. Let $dom(M)$ denote the domain of the map M. Let M, $M' \in \mathbb{M}$. We have

1. $[\,] \rhd M$
2. $M \rhd M' \Leftrightarrow dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ M(a) \rhd M'(a)$

In what follows, we present some utility functions that we use in the elaboration of our semantic model. Given two maps M and M', we write $M \dagger M'$ for the overwriting of the map M by the associations of the map M', i.e. the domain of $M \dagger M'$ is $dom(M) \cup dom(M')$ and we have

$$(M \dagger M')(a) \quad = \quad \begin{cases} M'(a), & \text{if } a \in dom(M'); \\ M(a), & \text{Otherwise.} \end{cases}$$

We use a tuple projection function π_n, which selects the element at position n in a tuple. For a pair $z = (x, y)$, the first element is denoted by $fst(z)$ and the second by $snd(z)$.

We also define a function φ that computes the elements of a map.

$$\begin{array}{lcl} \varphi & : & \mathbb{M} \rightarrow \mathcal{P}(\mathcal{P}_f(V) \times V) \text{ defined by} \\ \varphi(M) & = & \begin{cases} \emptyset, & \text{if } M = [\,]; \\ \bigcup_{a \in dom(M)} \{a\} \ \cup \ \varphi(M(a)) \ , & \text{Otherwise.} \end{cases} \end{array}$$

Moreover, we define a function $\mathcal{D}$ that computes the dependence relation between the elements of a map as follows:

$$\begin{array}{lcl} \mathcal{D} & : & \mathcal{P}(\mathcal{P}_f(V) \times V) \rightarrow \mathcal{P}(V \times V) \text{ defined by} \\ \mathcal{D}(S) & = & \{(e, \pi_2(a)) |\ a \in S \wedge e \in \pi_1(a)\} \end{array}$$

We define $\mathbb{T} \subseteq \mathbb{M}$ to be the space of dependence maps such that

$M \in \mathbb{T}$ if the reflexive transitive closure of $\mathcal{D}(\varphi(M))$ is a partial order relation.

This condition states that we have no cycles in any element of $\mathbb{T}$. It is the first *healthiness condition* in our semantic model. Moreover, the domain of each element of $\mathbb{T}$ should contain just *initials* (an initial is an event having an empty set of predecessors), i.e. $\forall M \in \mathbb{T}.\ \forall e \in dom(M).\ \pi_1(e) = \emptyset$.

Example of a Dependence Map

In what follows, we give an illustration of a dependence map as well as its graphical representation.

$$\begin{tikzcd}
a_1 \arrow[d] & & b_1 \arrow[d] & \\
c_1 \arrow[d] \arrow[dr] & & d_1 \arrow[dl] & \\
d_2 & b_2 \arrow[d] \arrow[dr] & & e_1 \arrow[dl] \\
& a_2 & f_1 \arrow[d] & \\
& & g_1 &
\end{tikzcd}$$

Figure 7.1: Example of a graphical representation of a dependence map

$$\begin{array}{lclclclclcl}
[(\emptyset,a_1) & \mapsto & [(\{a_1\},c_1) & \mapsto & [(\{c_1\},d_2) & \mapsto & [\,], & & & & \\
 & & & & (\{c_1,d_1\},b_2) & \mapsto & [(\{b_2\},a_2) & \mapsto & [\,], & & \\
 & & & & & & (\{b_2,e_1\},f_1) & \mapsto & [(\{f_1\},g_1) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &], & & & & & & & & \\
(\emptyset,b_1) & \mapsto & [(\{b_1\},d_1) & \mapsto & [(\{c_1,d_1\},b_2) & \mapsto & [(\{b_2\},a_2) & \mapsto & [\,], & & \\
 & & & & & & (\{b_2,e_1\},f_1) & \mapsto & [(\{f_1\},g_1) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &], & & & & & & & & \\
(\emptyset,e_1) & \mapsto & [(\{b_2,e_1\},f_1) & \mapsto & [(\{f_1\},g_1) & \mapsto & [\,] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & & \\
] & & & & & & & & & &
\end{array}$$

Figure 7.1 outlines the graphical representation of this dependence map.

7.3.2 Labelled Dependence Maps

The space of labelled dependence maps, which we call $\mathbb{R}$, contains dependence maps with their labelling functions. A labelling function associates each event of a dependence map with an action. The space $\mathbb{R}$ is defined as follows:

$$\mathbb{R} \;=\; \mathbb{T} \times (V \overset{\sim}{\longrightarrow} \Sigma)$$

Note that we will use the set notation to encode this labelling function. In fact, this notation make the proofs simple to do.

Hereafter, we present an example of a labelled dependence map (M, λ).

$$\begin{array}{lcll} M & = & [(\emptyset,a_1) & \mapsto \; [(\{a_1\},b_1) \mapsto [\,], \\ & & & \quad\;\; (\{a_1\},c_1) \mapsto [\,] \\ & & & \quad\;\;] \\ & &] & \end{array}$$

The labelling of the dependence map M is the following: $\lambda = \{(a_1, a), (b_1, b), (c_1, c)\}$.

We define the events set of an element of $\mathbb{R}$ by the function:

$$\begin{array}{lcl} \xi & : & \mathbb{R} \to \mathcal{P}(V) \text{ where} \\ \xi((M, \lambda)) & = & dom(\lambda) \end{array}$$

It is worth to mention that a labelled dependence map is considered as finite if its event set is finite.

7.3.3 Deterministic Processes

The space of deterministic processes, which we call $\mathbb{C}$, is a set of pairs. Each pair is composed of a labelled dependence map together with the resources needed by this map in the future. These resources are called continuation resources. More precisely, the dependence map represents what is already observed. It can evolve to any dependence map provided that any new executed action uses a subset of the continuation resources as it will be shown when we specify the ordering over the space $\mathbb{C}$.

More accurately, let $\mathcal{R}$ be a finite set of resources. The space $\mathbb{C}$ is defined as follows:

$$\mathbb{C} \;=\; \mathbb{R} \times \mathcal{P}(\mathcal{R})$$

A process $p = (r_p, R_p) \in \mathbb{C}$ such that $r_p = (M_p, \lambda_p)$ is considered as finite if and only if r_p is finite. The characterization of finite processes will be useful in determining the compact elements of the space $\mathbb{C}$.

To give meaning to recursion, we have to endow the space $\mathbb{C}$ with an ordering. To elaborate this ordering, we have to provide some definitions in what follows. We suppose that we have a function $res : \Sigma \to \mathcal{P}(\mathcal{R})$, which associates an action with a resource set. Note that we assume that each action uses a non-empty set of resources. To lighten the notation, we use $\hat{a}$ to denote the resource set needed by an action a, i.e. $\hat{a} = res(a)$.

The definition of this function is extended to $\mathcal{P}(\Sigma)$ as follows:

$$\begin{array}{lcl} res & : & \mathcal{P}(\Sigma) \to \mathcal{P}(\mathcal{R}) \\ res(A) & = & \bigcup_{a \in A} res(a) \end{array}$$

The definition of this function is also extended to the space of labelled dependence maps $\mathbb{R}$ as follows:

$$\begin{array}{lcl} res & : & \mathbb{R} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ res((M,\lambda)) & = & \bigcup_{e \in \xi((M,\lambda))} res(\lambda(e)) \end{array}$$

Finally, the definition of the function res is extended to the space deterministic processes $\mathbb{C}$ as follows:

$$\begin{array}{lcl} res & : & \mathbb{C} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ res(p) & = & res(r_p) \cup R_p \end{array}$$

The last extension of the function aims to compute the resources needed by a deterministic process. These resources are those that are already used together with those that are needed for the continuation of the process, i.e. for the future.

Now, we are ready to define the ordering $\sqsubseteq_{\mathbb{C}}$ over the space $\mathbb{C}$. Let $p, q \in \mathbb{C}$, p is denoted by (r_p, R_p) where $r_p = (M_p, \lambda_p)$ and q by (r_q, R_q) where $r_q = (M_q, \lambda_q)$. We have

$$p \sqsubseteq_{\mathbb{C}} q \quad \Leftrightarrow \quad r_p \preceq r_q \wedge R_p \supseteq R_q \cup res(r_p^{-1} r_q) \quad \text{where}$$

$$r_p \preceq r_q \quad \Leftrightarrow \quad M_p \rhd M_q \wedge \lambda_p \subseteq \lambda_q$$

$$r_p^{-1} r_q \quad = \quad (M_q \backslash M_p, \lambda_q \setminus \lambda_p)$$

$$M' \backslash M \quad = \quad \begin{cases} [(\emptyset, \pi_2(b)) \mapsto M'(b) \mid b \in dom(M')], & \text{if } M = [\ ]; \\ [a \mapsto M'(a) \mid a \in dom(M') \setminus dom(M)] \dagger & \\ \dagger_{a \in dom(M) \cap dom(M')} M'(a) \backslash M(a)\ , & \text{Otherwise.} \end{cases}$$

The ordering over $\mathbb{C}$ means that any process p can evolve to another process q with the constraint that only the resources specified in R_p are used. We call the constraint on the continuation resources that figure in the ordering $\sqsubseteq_{\mathbb{C}}$: *the resource constraint.* Note also that if $R_p = \emptyset$, the process p is finite and maximal with respect to the ordering $\sqsubseteq_{\mathbb{C}}$. Otherwise, it is a non-terminated process. It is worth to mention that the operator $\dagger$, which is already defined in Section 7.3.1, is used in a symmetric way since there are no cycles in our coding of dependence maps neither auto concurrency in our semantic model (recall that any action uses a non-empty set of resources).

Hereafter, we illustrate the operator $\setminus$ on dependence maps.

$$\begin{tikzcd}
e \arrow[r] & f & & & & & & e \arrow[r] & f \\
b \arrow[dr] & & & & b \arrow[dr] & & & & \\
& c \arrow[r] & d & \setminus & & c & = & d & \\
a \arrow[ur] & & & & a \arrow[ur] & & & &
\end{tikzcd}$$

Figure 7.2: An example illustrating the operator $\setminus$ on dependence maps

Let

$$\begin{array}{lllllllll}
M' & = & [(\emptyset,a) & \mapsto & [(\{a,b\},c) & \mapsto & [(\{c\},d) & \mapsto & [\,] \\
 & & & & & &] & & \\
 & & & &], & & & & \\
 & & (\emptyset,b) & \mapsto & [(\{a,b\},c) & \mapsto & [(\{c\},d) & \mapsto & [\,] \\
 & & & & & &] & & \\
 & & & &], & & & & \\
 & & (\emptyset,e) & \mapsto & [(\{e\},f) & \mapsto & [\,] & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}$$

$$\begin{array}{lllllll}
M & = & [(\emptyset,a) & \mapsto & [(\{a,b\},c) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (\emptyset,b) & \mapsto & [(\{a,b\},c) & \mapsto & [\,] \\
 & & & &] & & \\
 & &] & & & &
\end{array}$$

We have

$$\begin{array}{lllllll}
M'\backslash M & = & [(\emptyset,d) & \mapsto & [\,], & & \\
 & & (\emptyset,e) & \mapsto & [(\{e\},f) & \mapsto & [\,] \\
 & & & &] & & \\
 & &] & & & &
\end{array}$$

This example is graphically outlined in Figure 7.2.

In what follows, we provide some results related to the extension of the function res over the space $\mathbb{C}$. These results are needed to establish the algebraic properties of our process space as well as the monotonicity of our semantic functions.

If $r \preceq r'$, then we have

$$(7.1) \quad \xi(r^{-1}r') \quad = \quad \xi(r') \setminus \xi(r)$$

Moreover, in this particular case we have

$$(7.2) \quad res(r^{-1}r') \quad = \quad \emptyset \quad \Leftrightarrow \quad r = r'$$

Note also that if $r \preceq s \preceq t$, then we have

$$(7.3) \quad res(r^{-1}t) \quad = \quad res(r^{-1}s) \cup res(s^{-1}t)$$

Moreover for $p \sqsubseteq_{\mathbb{C}} p'$, we have

$$(7.4) \quad res(p) \quad \supseteq \quad res(p')$$

The proofs of these results can be easily established using the relation between the function ξ and the labelling function λ.

We also need the following definition: $\forall p, q \in \mathbb{C}$. if $r_p \preceq r_q$, then

$$p^{-1}q \quad = \quad (r_p^{-1}r_q, R_q)$$

This definition is needed to establish the proofs, which are related to the algebraic properties of our process space.

We also have the following constraint over the elements of $\mathbb{C}$: $\forall p \in \mathbb{C}$. $resinf(r_p) \subseteq R_p$, where $resinf(r_p)$ represents the resources required by the actions that occur infinitely often in the labelled dependence map r_p and which is defined as follows:

$$\begin{array}{lcl} resinf & : & \mathbb{R} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ resinf(x) & = & \bigcap\{res(r^{-1}x), \ r \text{ finite and } r \preceq x\} \end{array}$$

This condition means that all the needed resources for the continuation are specified in R_p. This is the second *healthiness condition* in our model. In addition, this condition is needed for establishing the algebraicity of the space $\mathbb{C}$.

Hereafter, we provide a result that is useful for establishing the algebraic properties of our process space.

For $r \preceq t$, we have:

$$(7.5) \quad resinf(t) \subseteq resinf(r) \cup res(r^{-1}t)$$

The proof of this result is straightforward.

Examples

As examples of deterministic processes, we provide the following pairs: $(([\,], \emptyset), \emptyset)$ and $(([\,], \emptyset), \mathcal{R})$. The first represents a process that successfully terminates while the second denotes a process that cannot evolve since it is requesting all the resources. The latter represents the least element of the space $\mathbb{C}$.

7.3.4 Space of Non-Deterministic Processes

In what follows, we present the space of non-deterministic processes, which we call $\mathbb{D}$. This space is the set of denotations that are associated with each syntactic term of our language $\mathcal{L}$.

The space $\mathbb{D}$ is defined as the set of non-empty upper sets that are subsets of $\mathbb{C}$:

$$\begin{array}{lcl} \mathbb{D} & = & \{X \subseteq \mathbb{C} \mid X \neq \emptyset \wedge X = \uparrow X\} \text{ where} \\ \uparrow X & = & \bigcup_{x \in X} \uparrow x \text{ and } \uparrow x = \{y \in \mathbb{C} \mid x \sqsubseteq_{\mathbb{C}} y\} \end{array}$$

The space $\mathbb{D}$ is endowed with an ordering $\sqsubseteq_{\mathbb{D}}$ and defined as follows:

$$\begin{array}{lcl} P \sqsubseteq_{\mathbb{D}} Q & \Leftrightarrow & P \supseteq Q \\ & & \\ & \Leftrightarrow & \forall q \in Q.\ \exists p \in P.\ p \sqsubseteq_{\mathbb{C}} q \end{array}$$

The definition of the function res is extended to the space of non-deterministic processes $\mathbb{D}$ as follows:

$$\begin{array}{lcl} res & : & \mathbb{D} \rightarrow \mathcal{P}(\mathcal{R}) \text{ defined by} \\ res(P) & = & \bigcup_{p \in P} res(p) \end{array}$$

This extension is useful for establishing the resources needed by a recursive process as shown in Section 7.5.7.

Example

As an example of a non-deterministic process, we have $\uparrow(([\,], \emptyset), \mathcal{R})$, which denotes the deadlock process. This is the least element of the space $\mathbb{D}$.

7.4 Algebraic Properties

In order to give meaning to recursion, we have to establish some results about our process space. For the sake of clarity, the proofs are provided in Section 7.7.

The following proposition establishes the algebraicity of the space $\mathbb{C}$ and specifies its compact elements.

Proposition 7.4.1 ($\mathbb{C}$,$\sqsubseteq_{\mathbb{C}}$) *is algebraic. The compact elements of* $\mathbb{C}$ *are processes having a finite events set.*

The following theorem is useful for proving that the space of non-deterministic processes, $\mathbb{D}$, is a local cpo.

Theorem 7.4.2 (Mislove[88])
If P is an algebraic domain, then $P_{US}(P) = \{X \subseteq P \mid \emptyset \neq X \wedge X = \uparrow X\}$ with $X + Y = X \cup Y$ and $X \sqsubseteq Y$ iff $Y \subseteq X$ is a local cpo.

The following corollary establishes an important result about the space $\mathbb{D}$. This result is important for establishing the existence of a fixed point for recursion.

Corollary 7.4.3 $(\mathbb{D},\sqsubseteq_{\mathbb{D}})$ *is a local cpo.*

The corollary is a direct result of the theorem 7.4.2, which states that a space of upper sets that are subsets of an algebraic domain is a local cpo. Hence, we deduce directly that $(\mathbb{D},\sqsubseteq_{\mathbb{D}})$ is a local cpo since $(\mathbb{C},\sqsubseteq_{\mathbb{C}})$ is an algebraic domain.

7.5 Semantics

In what follows, we present the denotational semantics of our language $\mathcal{L}$.

7.5.1 Semantic Interpretation Function

Let

- ω be a syntactic operator of our language $\mathcal{L}$,
- $\theta : \zeta \to \mathbb{D}$ be an environment, which associates a process variable with a non-deterministic process and
- $\tilde{\omega}^{+}$: $\mathbb{D} \times \mathbb{D} \to \mathbb{D}$ be the extension of the function $\tilde{\omega}$, which is the semantic function that operates on deterministic processes (elements of $\mathbb{C}$). The signature of $\tilde{\omega}$ is specified later since it depends on the syntactic operator.

The semantic interpretation function takes a syntactic term and returns a function that takes an environment mapping variables to non-deterministic processes and which returns a non-deterministic process (element of $\mathbb{D}$). More accurately, the semantic interpretation function is defined as follows:

$[\![\_]\!]_{\_} : \mathcal{L} \to \theta \to \mathbb{D}$

In the case of a binary syntactic language operator ω, the semantics is defined as follows:

$$\begin{array}{lcl} [\![\omega(P,Q)]\!](\theta) & = & \tilde{\omega}^{+}([\![P]\!](\theta),[\![Q]\!](\theta)) \\ & = & \bigcup \uparrow \tilde{\omega}(p,q) \text{ such that } p \in [\![P]\!](\theta) \wedge q \in [\![Q]\!](\theta) \end{array}$$

If the operator ω is unary, we have

$$\begin{array}{lcl} [\![\omega^{+}(P)]\!](\theta) & = & \tilde{\omega}^{+}([\![P]\!](\theta)) \\ & = & \bigcup \uparrow \tilde{\omega}(p) \text{ such that } p \in [\![P]\!](\theta) \end{array}$$

For the elaboration of the semantics of strict sequential composition, weak sequential composition, parallel composition and hiding, we provide the specification of the semantic function $\tilde{\omega}$. Note that we use the same notation for the language operators and their corresponding semantic operators. Moreover, to lighten the notation, we use $\sqsubseteq$ instead of $\sqsubseteq_{\mathbb{D}}$ and $\sqsubseteq_{\mathbb{C}}$ for the specification of these semantic operators. The appropriate notations can be deduced from the context.

The semantics of SKIP, STOP, a simple action and a variable are defined as follows:

$$\begin{array}{lcl} [\![\texttt{SKIP}]\!](\theta) & = & \uparrow (([\,],\emptyset),\emptyset) \\ & = & \{(([\,],\emptyset),\emptyset)\} \end{array}$$

$$[\![\texttt{STOP}]\!](\theta) = \uparrow (([\,],\emptyset),\mathcal{R})$$

$$\begin{array}{lcl} [\![\mathrm{a}]\!](\theta) & = & \uparrow (([(\emptyset,a_1) \mapsto [\,]],\{(a_1,\mathrm{a})\}),\emptyset) \\ & = & \{(([(\emptyset,a_1) \mapsto [\,]],\{(a_1,\mathrm{a})\}),\emptyset)\} \text{ where } a_1 \text{ is an event instance of action a} \end{array}$$

$$[\![X]\!](\theta) = \theta(X)$$

In what follows, we provide the semantic interpretation of our syntactic operators. For the strict sequential, weak sequential and parallel composition, we suppose that composed processes p and q are labelled differently, i.e. $dom(\lambda_p) \cap dom(\lambda_q) = \emptyset$.

7.5.2 Non-Deterministic Choice

The choice semantics is the following:

$$[\![\oplus_{i\in I} P_i]\!](\theta) = \bigcup_{i\in I}[\![P_i]\!](\theta)$$

There are three known semantic strategies for dealing with non-determinism:

- Angelic: Deadlock is avoided, which means that if there is one branch that does not deadlock, the whole process can progress.
- Demonic: Deadlock is catastrophic, which means that if one branch deadlocks, the whole process deadlocks.
- Erratic: The choice is arbitrary such that it can progress or deadlock.

For JVML/CLDC, if a deadlock happens in one branch, the sum process deadlocks. It is clear, from the above equation, that our semantics is demonic since if there is a deadlock in one branch, the sum deadlocks (recall that the semantics of `STOP` is $\uparrow (([\,], \emptyset), \mathcal{R})$).

7.5.3 Strict Sequential Composition

By strict sequential composition, we mean that the left-process is executed before the right-process. When the left-process needs resources for its continuation, the second process is not executed.

We define the function *Terminals*, which computes the set of terminal events of a finite map as follows:

$$Terminals \quad : \quad \mathbb{M} \rightarrow \mathcal{P}(V)$$

$$Terminals(M) \quad = \quad \begin{cases} \emptyset, & \text{if } M = [\,]; \\ \bigcup_{a \in dom(M)} \{\pi_2(a) \mid M(a) = [\,]\} \cup & \\ Terminals(M(a)), & \text{Otherwise.} \end{cases}$$

Let $\sigma_M : \mathcal{P}_f(V) \times V \rightarrow \mathcal{P}_f(V) \times V$ be a family of substitutions that update the past of an event by the terminals of the map M. This family is defined by

$$\sigma_M(b) \quad = \quad (Terminals(M), \pi_2(b))$$

We also define a substitution σ over a map M. This substitution is applied just on the initials of M. It will be used to update the right-hand process by the terminals of the dependence map of the left-hand process.

$$M\sigma \quad = \quad [\sigma(a) \mapsto M(a) \mid a \ \in \ dom(M)]$$

Now, we are ready to define the strict sequential composition semantics.

$$; \quad : \quad \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$$

$$p\ ;\ q \quad = \quad \begin{cases} ((M_p, \lambda_p), R_p \cup res(q)), & \text{if } R_p \neq \emptyset; \\ ((S(M_p, M_q), \lambda_p \ \cup \ \lambda_q), R_q), & \text{Otherwise.} \end{cases}$$

Where:

$$S \quad : \quad \mathbb{M} \times \mathbb{M} \rightarrow \mathbb{M}$$

$$S(M, M') = \begin{cases} M', & \text{if } M = [\,]; \\ [a \mapsto S(M(a), M') \mid a \in dom(M) \ \wedge \ M(a) \neq [\,]] \dagger & \\ [a \mapsto M'\sigma_M \mid a \in dom(M) \ \wedge \ M(a) = [\,]], & \text{Otherwise.} \end{cases}$$

Examples:

For all the examples that we provide, we adopt the following notation to make the presentation simple and clear: we write $a \ D \ b$ if the actions a and b are dependent, i.e. $res(a) \cap res(b) \neq \emptyset$. Otherwise, we say that a and b are independent. Note also that an event a_1 is denoted by a when there is no need to tag the associated action.

In what follows, two examples are provided in order to illustrate the strict sequential composition.

Example 1: In the first example, we outline the sequential composition between two finite processes p and q. Note that these processes are finite since their continuation resource sets are empty. This makes the continuation resource set of the process, that is the result of the composition, equal to the empty set.

$$\begin{array}{lcl} M_p & = & [(\emptyset,a) \ \mapsto \ [(\{a\},b) \mapsto [\,], \\ & & \qquad\qquad\quad (\{a\},c) \mapsto [\,] \\ & & \qquad\qquad\quad] \\ & &] \end{array}$$

$$\begin{array}{lcl} \lambda_p & = & \{(a,a),(b,b),(c,c)\} \\ R_p & = & \emptyset \end{array}$$

$$\begin{array}{lcl} M_q & = & [(\emptyset,d) \ \mapsto \ [(\{d\},e) \mapsto [\,], \\ & & \qquad\qquad\quad (\{d\},f) \mapsto [\,] \\ & & \qquad\qquad\quad] \\ & &] \end{array}$$

$$\begin{array}{lcl} \lambda_q & = & \{(d,d),(e,e),(f,f)\} \\ R_q & = & \emptyset \end{array}$$

$$Terminals(M_p) \ = \ \{b,c\}$$

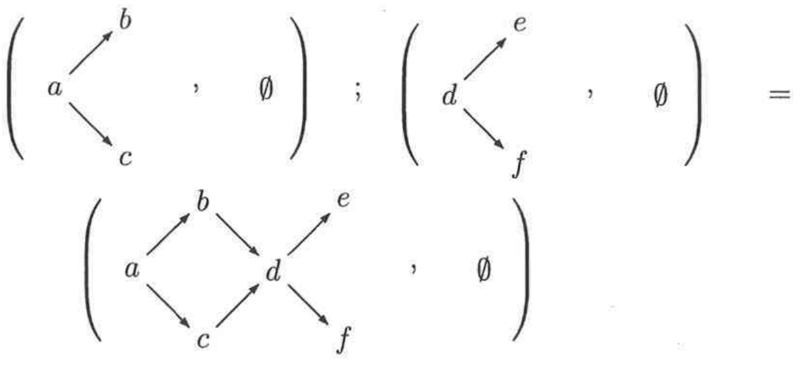


Figure 7.3: A strict sequential composition of two finite processes

$$\begin{array}{lllllllllll}
M_{p;q} & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b,c\},d) & \mapsto & [(\{d\},e) & \mapsto & [\,], \\
 & & & & & & & & (\{d\},f) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &], & & & & \\
 & & & & (\{a\},c) & \mapsto & [(\{b,c\},d) & \mapsto & [(\{d\},e) & \mapsto & [\,], \\
 & & & & & & & & (\{d\},f) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & &
\end{array}$$

$$\begin{array}{lll}
\lambda_{p;q} & = & \lambda_p \cup \lambda_q \\
R_{p;q} & = & \emptyset
\end{array}$$

Figure 7.3 outlines the sequential composition of these processes.

Example 2: In the second example, we outline the sequential composition of a finite process p and an infinite process q. The continuation resources set of the process, that is result of the composition, is equal to the continuation resources set of the process q.

$$\begin{array}{lllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) \;\mapsto\; [\,] \\
 & & & &] \\
 & &] & &
\end{array}$$

$$\begin{array}{lll}
\lambda_p & = & \{(a,a),(b,b)\} \\
R_p & = & \emptyset
\end{array}$$

$$\begin{array}{lllllllll}
M_q & = & [(\emptyset,a) & \mapsto & [(\{a\},c_1) & \mapsto & [(\{c_1\},c_2) & \mapsto & [(\{c_2\},c_3) \;\mapsto\; \ldots \\
 & & & & & & & &] \\
 & & & & & &] & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}$$

$$\left(a \longrightarrow b \;,\; \emptyset \right) \;;\; \left(a \longrightarrow c_1 \longrightarrow c_2 \;\cdot\;\cdot\;\cdot\;\cdot \;,\; \hat{c} \right) \;=$$

$$\left(a_1 \longrightarrow b \longrightarrow a_2 \longrightarrow c_1 \longrightarrow c_2 \;\cdot\;\cdot\;\cdot\;\cdot \;,\; \hat{c} \right)$$

Figure 7.4: A strict sequential composition of a finite and an infinite Process

$$\begin{array}{lll} \lambda_q & = & \{(a,a),(c_1,c),(c_2,c),\ldots\} \\ R_q & = & \hat{c} \end{array}$$

$$M_{p;q} =$$

$$[(\emptyset,a_1) \mapsto [(\{a_1\},b) \mapsto [(\{b\},a_2) \mapsto [(\{a_2\},c_1) \mapsto [(\{c_1\},c_2) \mapsto [(\{c_2\},c_3) \mapsto \ldots]]]]]]$$

$$\begin{array}{lll} \lambda_{p;q} & = & \lambda_p \cup \lambda_q \\ R_{p;q} & = & \hat{c} \end{array}$$

Figure 7.4 outlines the sequential composition of the processes, which are provided in the second example.

7.5.4 Hiding

By hiding, we mean that the events using a subset of some resource set become unobservable. These events are replaced by events that are instances of invisible actions.

Let R be a finite set of resources and $V_R^p = \{e \in \xi(r_p) \mid res(\lambda_p(e)) \subseteq R\}$ be the set of events in p that use resources in R.

Let $\eta : Act \to Act_\tau$ be an injective relabelling function that associates an action a with an invisible action τ such that $\eta(\text{a}) = \tau \Rightarrow res(\text{a}) = res(\tau)$.

Let $\sigma : V \to V$ be an injective relabelling function that associates an event e with a fresh instance of an invisible action if e uses a subset of R. More precisely, σ is defined as follows:

$$\sigma(e) = \begin{cases} (\eta(fst(e)), snd(e)), & \text{if } res(fst(e)) \subseteq R; \\ e, & \text{Otherwise.} \end{cases}$$

Moreover, we denote by σ^+ the extension of σ over event sets. σ^+ is defined as follows:

$$\begin{array}{lcl} \sigma^+ & : & \mathcal{P}(V) \rightarrow \mathcal{P}(V) \\ \sigma^+(S) & = & \{\sigma(e) \mid e \in S\} \end{array}$$

Now, we are ready to provide the hiding semantics.

$$\begin{array}{lcl} \backslash & : & \mathbb{C} \times \mathcal{P}(\mathcal{R}) \rightarrow \mathbb{C} \\ & & \\ p\backslash R & = & (r_p, R_p)\backslash R \\ & = & (r_p\backslash R, R_p) \\ & & \\ r_p\backslash R & = & (H_R(M_p), \lambda_{p\backslash R}) \\ & & \\ H_R & : & \mathbb{M} \rightarrow \mathbb{M} \\ H_R(M_p) & = & \begin{cases} [\,], & \text{if } M_p = [\,]; \\ [(\sigma^+(\pi_1(a)), \sigma(\pi_2(a))) \mapsto H_R(M_p(a)) \mid a \in dom(M_p)], & \text{Otherwise.} \end{cases} \\ & & \\ \lambda_{p\backslash R} & = & (\lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}) \cup \{(\sigma(e), \eta(\lambda_p(e))) \mid e \in V_R^p\} \end{array}$$

Note that when the hiding is applied on a process, it is not penalized at the level of continuation resources. This is in order to ensure monotonicity of hiding.

Examples:

In what follows, we provide two examples that illustrate the semantics of hiding in our model.

Example 1: The following example shows the hiding of events using the resource set $\hat{b}$ from a process p. We suppose that $res(\lambda_p(a)) \not\subseteq \hat{b}$, $res(\lambda_p(c)) \not\subseteq \hat{b}$ and $res(\lambda_p(d)) \not\subseteq \hat{b}$. In this context, the events, which are instances of the action b become unobservable.

$$\begin{array}{lcl} M_p & = & [(\emptyset,a) \mapsto [(\{a\},b) \mapsto [(\{b\},c) \mapsto [(\{c\},d) \mapsto [\,] \\ & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \\ & & \qquad\qquad\qquad\qquad\qquad] \\ & & \qquad\qquad] \\ & &] \end{array}$$

$$\begin{array}{lcl} \lambda_p & = & \{(a,a),(b,b),(c,c),(d,d)\} \\ R_p & = & \emptyset \\ V_{\hat{b}}^p & = & \{b\} \end{array}$$

$$\left(a \longrightarrow b \longrightarrow c \longrightarrow d \;,\; \emptyset \right) \setminus \hat{b} \;=\; \left(a \longrightarrow \tau \longrightarrow c \longrightarrow d \;,\; \emptyset \right)$$

Figure 7.5: A hiding of events using some resource set from a process

$$\left(c_1 \longrightarrow c_2 \longrightarrow c_3 \;\cdot\;\cdot\;\cdot\;\cdot \;,\; \hat{c} \right) \;\setminus \hat{c} \;=\; \left(\tau_1 \longrightarrow \tau_2 \longrightarrow \tau_3 \;\cdot\;\cdot\;\cdot\;\cdot \;,\; \hat{c} \right)$$

Figure 7.6: A hiding of events using a resource set from a recursive process that results into a divergence

$$\begin{array}{lcl}
M_{p\setminus\hat{b}} & = & [(\emptyset,a) \;\mapsto\; [(\{a\},\tau) \;\mapsto\; [(\{\tau\},c) \;\mapsto\; [(\{c\},d) \;\mapsto\; [\,] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
 & & \qquad\qquad\qquad\quad] \\
 & & \quad] \\
\lambda_{p\setminus\hat{b}} & = & (\lambda_p \setminus \{(b,b)\}) \cup \{(\tau,\eta(b))\} \\
R_{p\setminus\hat{b}} & = & R_p
\end{array}$$

Figure 7.5 outlines graphically the first example.

Example 2: Figure 7.6 outlines another example of resource hiding. In this example, the hiding makes all the events of the hidden process unobservable. Besides, we see in this example that we have a divergence, i.e. the hidden process performs infinite invisible actions.

7.5.5 Weak Sequential Composition

A weak sequential composition means that the events of the composed processes can be executed in parallel when there is no dependence. If there is a dependence between an event e_1 in the left-hand process and another one e_2 in the right-hand process, e_1 is executed before e_2. For reasons of monotonicity, this operator allows the execution of the maximal prefix of the right-hand process that does not use the continuation resources of the left-hand one. The weak sequential composition is as follows:

$$\begin{array}{lcl}
\circ & : & \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C} \\[2ex]
p \circ q & = & (r_p \bullet r_{\mu^q_{R_p}}, R_p \cup R_q \cup res(r_{\mu^q_{R_p}}^{-1} r_q)) \text{ where} \\[2ex]
r_p \bullet r_{\mu^q_{R_p}} & = & (\Rsh_{\Phi_{p,\mu^q_{R_p}}} (M_p, \Lsh_{\Phi_{p,\mu^q_{R_p}}} (M_{\mu^q_{R_p}}, r_p)) \ddagger \Lsh_{\Phi_{p,\mu^q_{R_p}}} (M_{\mu^q_{R_p}}, r_p), \lambda_p \cup \lambda_{\mu^q_{R_p}}) \\[2ex]
\mu^q_{R_p} & = & \sqcup\{q' \sqsubseteq q \mid res(\pi_1(q')) \cap R_p = \emptyset\} \\[2ex]
\Phi_{p,\mu^q_{R_p}} & = & \{(e, e') \in \xi(r_p) \times \xi(r_{\mu^q_{R_p}}) \mid res(\lambda_p(e)) \cap res(\lambda_{\mu^q_{R_p}}(e')) \neq \emptyset\}
\end{array}$$

$$\Lsh_{\Phi} (M, r) = \begin{cases} [\,], & \text{if } M = [\,]; \\ [(\pi_1(b) \cup \{a \in \xi(r) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mapsto & \\ \Lsh_{\Phi} (M(b), r) \mid b \in dom(M)], & \text{Otherwise.} \end{cases}$$

$$\Rsh_{\Phi} (M, M') = \begin{cases} [\,], & \text{if } M = [\,]; \\ [a \mapsto Succ(a, M', \Phi) \dagger \Rsh_{\Phi} (M(a), M') \mid a \in dom(M)], & \text{Otherwise.} \end{cases}$$

$$Succ(a, M, \Phi) = \begin{cases} [\,], & \text{if } M = [\,]; \\ [b \mapsto M(b) \mid b \in dom(M) \wedge (\pi_2(a), \pi_2(b)) \in \Phi] \dagger & \\ \dagger_{b \in dom(M)} Succ(a, M(b), \Phi), & \text{Otherwise.} \end{cases}$$

$$M \ddagger M' = M \dagger [a \mapsto M'(a) \mid a \in dom(M') \wedge \pi_1(a) = \emptyset]$$

Note that

- $\Lsh_{\Phi} (M, r)$ updates the dependence map M by adding the events of the labelled dependence map r, which are predecessors of any of its events in the relation Φ.
- $\Rsh_{\Phi} (M, M')$ updates the dependence map M by adding the events of the dependence map M', which are successors of any of its events in the relation Φ. Particularly, $Succ(a, M', \Phi)$ computes the successors in M' of the event $\pi_2(a)$. These successors are deduced from the relation Φ.
- The operator $\ddagger$ takes two maps as arguments and computes a dependence map using these arguments, i.e. the computed map has a domain that contains only initials. This is due to the fact that the second argument of the operator $\ddagger$ is in general an element of $\mathbb{M}$ and not of $\mathbb{T}$.

Examples:

In the sequel, we present three examples that illustrate the weak sequential composition.

Example 1: The first example shows a case in which the maximal prefix of the right-hand process, that is not using the continuation resources of the left-hand, is computed in order to do the composition. In this case, the operator $\circ$ allows independent events to be executed in parallel while imposing an order between dependent events belonging to the composed processes. The order of the sequential composition is in favor of the left-hand process. To lighten the notation, we denote the set of direct predecessors of an event t before the composition by X_t.

$$
\begin{array}{lclllllllll}
p & = & ((M_p,\lambda_p),\hat{c}) \\
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b\},c_1) & \mapsto & [(\{c_1\},c_2) & \mapsto & [(\{c_2\},c_3) \mapsto \ldots \\
 & & & & & & & & & &] \\
 & & & & & & & &] \\
 & & & & & &] \\
 & & & &] \\
 & &] \\
\lambda_p & = & \{(a,a),(b,b),(c_1,c),(c_2,c),\ldots\} \\
q & = & ((M_q,\lambda_q),\emptyset) \\
M_q & = & [(\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto & [\,] \\
 & & & &], \\
 & & (\emptyset,f) & \mapsto & [\,] \\
 & &] \\
\lambda_q & = & \{(d,d),(e,e),(f,f)\}
\end{array}
$$

We suppose that $a\ D\ d$, $b\ D\ d$ and $c\ D\ e$.

$$
\begin{array}{lclll}
M_{\mu^q_{R_p}} & = & [(\emptyset,d) & \mapsto & [\,], \\
 & & (\emptyset,f) & \mapsto & [\,] \\
 & &]
\end{array}
$$

$$
\begin{array}{lcl}
\lambda_{\mu^q_{R_p}} & = & \{(d,d),(f,f)\} \\
R_{\mu^q_{R_p}} & = & \hat{e} \\
\Phi & = & \{(a,d),(b,d),(c,e)\} \\
r_p \bullet r_{\mu^q_{R_p}} & = & (\Rsh_\Phi(M_p,\Lsh_\Phi(M_{\mu^q_{R_p}},r_p)) \ddagger \Lsh_\Phi(M_{\mu^q_{R_p}},r_p),\lambda_p \cup \lambda_{\mu^q_{R_p}})
\end{array}
$$

$$
\begin{array}{lclll}
\Lsh_\Phi(M_{\mu^q_{R_p}},r_p) & = & [(X_d \cup \{a,b\},d) & \mapsto & [\,], \\
 & & (X_f,f) \mapsto [\,] \\
 & &]
\end{array}
$$

Let $X'_d{=}X_d \cup \{a,b\}$. We have

$$\Rsh_\Phi(M_p,\Lsh_\Phi(M_{\mu^q_{R_p}},r_p)) \quad =$$

$$\left(\begin{tikzcd} a \arrow[r] & b \arrow[r] & c_1 \arrow[r] & c_2 & \cdots , & \hat{c} \end{tikzcd} \right) \circ \left(\begin{tikzcd} d \arrow[r] & e , & \emptyset \\ f \end{tikzcd} \right) =$$

$$\left(\begin{tikzcd} a \arrow[r] \arrow[d] & b \arrow[r] \arrow[dl] & c_1 \arrow[r] & c_2 & \cdots , & \hat{c} \cup \hat{e} \\ d & f \end{tikzcd} \right)$$

Figure 7.7: Weak sequential composition of two processes

$$\begin{array}{lllllllll}
[(X_a,a) & \mapsto & [(X_b,b) & \mapsto & [(X_{c_1},c_1) & \mapsto & [(X_{c_2},c_2) & \mapsto & \ldots \\
 & & & & & &], & & \\
 & & & & (X'_d,d) & \mapsto & [\,] & & \\
 & & & &], & & & & \\
 & & (X'_d,d) & \mapsto & [\,] & & & & \\
 & &] & & & & & & \\
] & & & & & & & &
\end{array}$$

$$\begin{array}{lllllll}
r_p \bullet r_{\mu^q_{R_p}} & = & & & & & \\
[(X_a,a) & \mapsto & [(X_b,b) & \mapsto & [(X_{c_1},c_1) & \mapsto & [(X_{c_2},c_2) \mapsto \ldots \\
 & & & & & &], \\
 & & & & (X'_d,d) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (X'_d,d) & \mapsto & [\,] & & \\
 & &], & & & & \\
(X_f,f) & \mapsto & [\,] & & & & \\
] & & & & & &
\end{array}$$

$$p \circ q = (r_p \bullet r_{\mu^q_{R_p}}, \hat{c} \cup \hat{e})$$

Figure 7.7 outlines the weak sequential composition of the processes that are provided in the first example.

Example 2: The following second example shows how the operator $\circ$ allows parallel execution of the events of the composed processes if they are independent.

Hereafter, we suppose that a,b,d and e are independent.

$$\begin{array}{lllllllllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b\},c_1) & \mapsto & [(\{c_1\},c_2) & \mapsto & [(\{c_2\},c_3) \mapsto \ldots \\
 & & & & & & & & & &] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & &
\end{array}$$

$$\left(a \longrightarrow b \longrightarrow c_1 \longrightarrow c_2 \;\cdot\;\cdot\;\cdot\; , \; \hat{c} \right) \circ \left(d \longrightarrow e \longrightarrow c \; , \; \emptyset \right) =$$

$$\left(\begin{array}{l} a \longrightarrow b \longrightarrow c_1 \longrightarrow c_2 \;\cdot\;\cdot\;\cdot\;\cdot\; , \; \hat{c} \\ d \longrightarrow e \end{array} \right)$$

Figure 7.8: Another example of weak sequential composition of two processes

$$\begin{array}{lcl}
\lambda_p & = & \{(a,a),(b,b),(c_1,c),(c_2,c),\ldots\} \\
R_p & = & \hat{c} \\
M_q & = & [(\emptyset,d) \mapsto [(\{d\},e) \mapsto [(\{e\},c) \mapsto [\,] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
 & & \qquad\qquad\qquad\quad] \\
 & &] \\
\lambda_q & = & \{(d,d),(e,e),(c,c)\} \\
R_q & = & \emptyset \\
\Phi & = & \emptyset \\
\mu^q_{\hat{c}} & = & (([(\emptyset,d) \mapsto [(\{d\},e) \mapsto [\,]]],\lambda_{\mu^q_{\hat{c}}}),\hat{c}) \\
\lambda_{\mu^q_{\hat{c}}} & = & \{(d,d),(e,e)\} \\
M_{p\,\circ\,q} & = & [(\emptyset,d) \mapsto [(\{d\},e) \mapsto [\,] \\
 & & \qquad\qquad\quad], \\
 & & \;\;(\emptyset,a) \mapsto [(\{a\},b) \mapsto [(\{b\},c_1) \mapsto [(\{c_1\},c_2) \mapsto [(\{c_2\},c_3) \mapsto \ldots \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad] \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad] \\
 & & \qquad\qquad\qquad\qquad\qquad\quad] \\
 & & \qquad\qquad\quad] \\
 & &] \\
\lambda_{p\,\circ\,q} & = & \lambda_p \cup \lambda_{\mu^q_{\hat{c}}} \\
R_{p\,\circ\,q} & = & \hat{c}
\end{array}$$

Figure 7.8 outlines the weak sequential composition of the processes that are provided in the second example.

Example 3: The third example shows the emergence of transfinite dependencies. More precisely, the event a is mapped to a transfinite successors set. Hereafter, we suppose that $a\ D\ c$ and $a\ D\ b$.

$$
\begin{array}{lllllllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b_1) & \mapsto & [(\{a,b_1\},b_2) & \mapsto & [(\{a,b_2\},b_3) \mapsto \ldots \\
 & & & & & & & &] \\
 & & & & & &], & & \\
 & & & & (\{a,b_1\},b_2) & \mapsto & [(\{a,b_2\},b_3) & \mapsto & \ldots \\
 & & & & & &], & & \\
 & & & & \ldots & & & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}
$$

$$
\begin{array}{lll}
\lambda_p & = & \{(a,a),(b_1,b),(b_2,b),\ldots\} \\
R_p & = & \hat{b}
\end{array}
$$

$$
\begin{array}{lllllll}
M_q & = & [(\emptyset,c_1) & \mapsto & [(\{c_1\},c_2) & \mapsto & [(\{c_2\},c_3) \mapsto \ldots \\
 & & & & & &] \\
 & & & &] & & \\
 & &] & & & &
\end{array}
$$

$$
\begin{array}{lll}
\lambda_q & = & \{(c_1,c),(c_2,c),\ldots\} \\
R_q & = & \hat{c} \\
\Phi & = & \{(a,c_1),(a,c_2),(a,c_3),\ldots\} \\
\mu_{\hat{b}}^{q} & = & q
\end{array}
$$

$$
\begin{array}{lllllllll}
M_{p\,\circ\,q} & = & [(\emptyset,a) & \mapsto & [(\{a\},b_1) & \mapsto & [(\{a,b_1\},b_2) & \mapsto & [(\{a,b_2\},b_3) \mapsto \ldots \\
 & & & & & & & &] \\
 & & & & & &], & & \\
 & & & & (\{a,b_1\},b_2) & \mapsto & [(\{a,b_2\},b_3) & \mapsto & \ldots \\
 & & & & & &], & & \\
 & & & & \ldots, & & & & \\
 & & & & (\{a\},c_1) & \mapsto & [(\{a,c_1\},c_2) & \mapsto & [(\{a,c_2\},c_3) \mapsto \ldots \\
 & & & & & & & &] \\
 & & & & & &], & & \\
 & & & & (\{a,c_1\},c_2) & \mapsto & [(\{a,c_2\},c_3) & \mapsto & \ldots \\
 & & & & & &], & & \\
 & & & & \ldots & & & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}
$$

$$
\begin{array}{lll}
\lambda_{p\,\circ\,q} & = & \lambda_p \cup \lambda_q \\
R_{p\,\circ\,q} & = & \hat{b} \cup \hat{c}
\end{array}
$$

Figure 7.9 outlines the weak sequential composition of the processes provided in the third example.

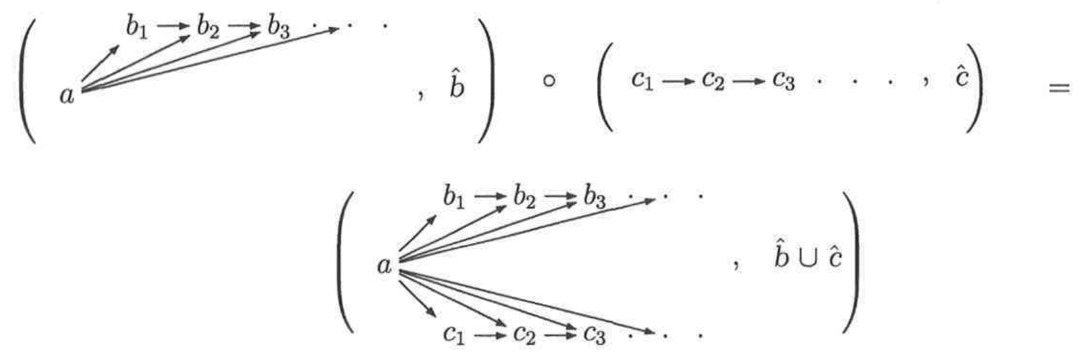


Figure 7.9: Example about the emergence of transfinite dependencies in the presence of weak sequential composition of two processes

7.5.6 Parallel Composition

The parallel composition of two processes p and q mixes true concurrency and non-determinism. Our semantics of parallelism introduces non-determinism only when two actions, sharing some resources, are executed in parallel. This allows to avoid state explosion, which is the main drawback of interleaving semantic models. The parallel composition semantics is as follows:

$$\| \quad : \quad \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{D}$$

$$p \parallel q = \begin{cases} \bigcup_{\Phi_i \in \Psi_{\mu^p_{R_q}, \mu^q_{R_p}}} \uparrow (r_{\mu^p_{R_q}} \|_{\Phi_i} r_{\mu^q_{R_p}}, R^p_q), & \text{if } R_{\mu^p_{R_q}} \cap res(r_{\mu^q_{R_p}}) = \emptyset \wedge \\ & \quad R_{\mu^q_{R_p}} \cap res(r_{\mu^p_{R_q}}) = \emptyset; \\ \uparrow (([\,], \emptyset), \mathcal{R}), & \text{Otherwise.} \end{cases}$$

Where:

$$r_{\mu^p_{R_q}} \|_{\Phi_i} r_{\mu^q_{R_p}} = (\looparrowright_{\Phi_i} (r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \bigcirc \looparrowright_{\Phi_i} (r_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}), \lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}})$$

$$R^p_q = R_p \cup R_q \cup res(r_{\mu^p_{R_q}}^{-1} r_p) \cup res(r_{\mu^q_{R_p}}^{-1} r_q)$$

$$\Psi_{\mu^p_{R_q}, \mu^q_{R_p}} = \{\Psi_i \cup \mathcal{D}(\varphi(M_{\mu^p_{R_q}})) \cup \mathcal{D}(\varphi(M_{\mu^q_{R_p}})) \mid \Psi_i \in \mathcal{D}_{\mu^p_{R_q}, \mu^q_{R_p}} \wedge \text{the reflexive transitive closure of } \Psi_i \cup \mathcal{D}(\varphi(M_{\mu^p_{R_q}})) \cup \mathcal{D}(\varphi(M_{\mu^q_{R_p}})) \text{ is a partial order relation}\}$$

$$\mathcal{D}_{\mu^p_{R_q},\mu^q_{R_p}} = \begin{cases} \{S \subseteq V_{\mu^p_{R_q},\mu^q_{R_p}} \cup V_{\mu^q_{R_p},\mu^p_{R_q}} \mid S \neq \emptyset \wedge \\ \ \forall (e,e') \in V_{\mu^p_{R_q},\mu^q_{R_p}} \cup V_{\mu^q_{R_p},\mu^p_{R_q}} . \\ \ (e,e') \in S \vee (e',e) \in S\}, & \text{if } V_{\mu^p_{R_q},\mu^q_{R_p}} \neq \emptyset \\ \\ \{\emptyset\}, & \text{Otherwise.} \end{cases}$$

$$V_{\mu^p_{R_q},\mu^q_{R_p}} = \{(e,e') \in \xi(r_{\mu^p_{R_q}}) \times \xi(r_{\mu^q_{R_p}}) \mid res(\lambda_{\mu^p_{R_q}}(e)) \cap res(\lambda_{\mu^q_{R_p}}(e')) \neq \emptyset\}$$

$$V_{\mu^q_{R_p},\mu^p_{R_q}} = \{(e,e') \in \xi(r_{\mu^q_{R_p}}) \times \xi(r_{\mu^p_{R_q}}) \mid res(\lambda_{\mu^q_{R_p}}(e)) \cap res(\lambda_{\mu^p_{R_q}}(e')) \neq \emptyset\}$$

$$\looparrowright_{\Phi} (r,r') = \begin{cases} [\,], & \text{if } M = [\,]; \\ [a \mapsto Successors(a, N_{\Phi}(r,r')) \mid \\ \ a \in dom(\Lsh_{\Phi} (\pi_1(r), r'))], & \text{Otherwise.} \end{cases}$$

$$\Lsh_{\Phi} (M,r) = \begin{cases} [\,], & \text{if } M = [\,]; \\ [(\pi_1(b) \cup \{a \in \xi(r) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mapsto \\ \Lsh_{\Phi} (M(b), r) \mid b \in dom(M)], & \text{Otherwise.} \end{cases}$$

$$N_{\Phi}(r,r') = \varphi(\Lsh_{\Phi} (\pi_1(r), r')) \cup \varphi(\Lsh_{\Phi} (\pi_1(r'), r))$$

$$Successors(a, N_{\Phi}(r,r')) = [b \mapsto Successors(b, N_{\Phi}(r,r')) \mid b \in N_{\Phi}(r,r') \wedge (\pi_2(a), \pi_2(b)) \in \Phi]$$

$$M \bigcirc M' = [a \mapsto M(a) \ \mid a \in dom(M) \wedge \pi_1(a) = \emptyset] \dagger [a \mapsto M'(a) \mid a \in dom(M') \wedge \pi_1(a) = \emptyset]$$

Note that

- $\Lsh_{\Phi} (M,r)$ updates the dependence map M by adding the events of the labelled dependence map r, which are predecessors of any of its events in the relation Φ.
- $\looparrowright_{\Phi} (r,r')$ updates the dependence map $\Lsh_{\Phi} (\pi_1(r), r')$ by adding the events of the labelled dependence map r', which are successors of any of its events in the relation Φ.
- The operator $\bigcirc$ computes, from its arguments, a dependence map, i.e. a map that has a domain containing only initials.
- For reasons of monotonicity, the maximal prefix of each of the composed processes, that does not use the continuation resources of the other one, is computed.

It is worth to mention that there are two main differences between the parallel and weak sequential composition. In fact, the parallel composition allows dependencies between the left and right-hand processes in both directions, while the weak sequential composition allows the dependencies only from the left to the right-hand process. Another major difference is that the parallel composition is non-deterministic while the weak sequential is deterministic (we speak about compositions over deterministic processes, i.e. elements of $\mathbb{C}$).

Examples:

In what follows, three examples are provided in order to illustrate the parallel composition.

Example 1: The first example shows that the parallel composition of two processes allows parallel execution of independent events while imposing a sequential order between dependent events. The sequential composition is not in favor of one of the composed processes.

$$
\begin{array}{lclllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b\},c) \quad \mapsto \quad [\,] \\
 & & & & & &] \\
 & & & &] & & \\
 & &] & & & &
\end{array}
$$

$$
\begin{array}{lcl}
\lambda_p & = & \{(a,a),(b,b),(c,c)\} \\
R_p & = & \emptyset
\end{array}
$$

$$
\begin{array}{lclllll}
M_q & = & [(\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto & [\,] \\
 & & & &] & & \\
 & &] & & & &
\end{array}
$$

$$
\begin{array}{lcl}
\lambda_q & = & \{(d,d),(e,e)\} \\
R_q & = & \emptyset \\
r_{\mu^p_{R_q}} & = & r_p \\
r_{\mu^q_{R_p}} & = & r_q
\end{array}
$$

We suppose that $a\ D\ e$ and $b\ D\ d$.

The possible dependence sets between p and q are

$$
\begin{array}{lcl}
\Phi_1 & = & \{(a,e),(b,d),(a,b),(b,c),(d,e)\} \\
\Phi_2 & = & \{(e,a),(d,b),(a,b),(b,c),(d,e)\} \\
\Phi_3 & = & \{(a,e),(d,b),(a,b),(b,c),(d,e)\}
\end{array}
$$

$$
r_p \parallel_{\Phi_1} r_q \quad = \quad (\looparrowright_{\Phi_1}(r_p,r_q) \bigcirc \looparrowright_{\Phi_1}(r_q,r_p),\lambda_p \cup \lambda_q)
$$

$$\Lsh_{\Phi_1}(M_p,r_q) = [(X_a,a) \mapsto [(X_b,b) \mapsto [(X_c,c) \mapsto [\,]\,]\,]\,]$$

$$\Lsh_{\Phi_1}(M_q,r_p) = [(X_d \cup \{b\},d) \mapsto [(X_e \cup \{a\},e) \mapsto [\,]\,]\,]$$

$$\begin{aligned} N_{\Phi_1}(r_p,r_q) &= N_{\Phi_1}(r_q,r_p) \\ &= \{(X_a,a),(X_b,b),(X_c,c),(X_d \cup \{b\},d),(X_e \cup \{a\},e)\} \end{aligned}$$

$$\begin{array}{l} \looparrowright_{\Phi_1}(r_p,r_q) = \\ \begin{array}{lllllllll} [(X_a,a) & \mapsto & [(X_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,], & & \\ & & & & (X_d \cup \{b\},d) & \mapsto & [(X_e \cup \{a\},e) & \mapsto & [\,] \\ & & & & & &] & & \\ & & & &], & & & & \\ & & (X_e \cup \{a\},e) & \mapsto & [\,] & & & & \\ & &]\ ] & & & & & & \end{array} \end{array}$$

$$\looparrowright_{\Phi_1}(r_q,r_p) = [(X_d \cup \{b\},d) \mapsto [(X_e \cup \{a\},e) \mapsto [\,]\,]\,]$$

$$\begin{array}{l} r_p \parallel_{\Phi_1} r_q = \\ \begin{array}{lllllllll} ([(X_a,a) & \mapsto & [(X_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,], & & \\ & & & & (X_d \cup \{b\},d) & \mapsto & [(X_e \cup \{a\},e) & \mapsto & [\,] \\ & & & & & &] & & \\ & & & &], & & & & \\ & & (X_e \cup \{a\},e) & \mapsto & [\,] & & & & \\ & &] & & & & & & \\ ],\lambda_p \cup \lambda_q) & & & & & & & & \end{array} \end{array}$$

$$r_p \parallel_{\Phi_2} r_q = (\looparrowright_{\Phi_2}(r_p,r_q) \bigcirc \looparrowright_{\Phi_2}(r_q,r_p),\lambda_p \cup \lambda_q)$$

$$\Lsh_{\Phi_2}(M_p,r_q) = [(X_a \cup \{e\},a) \mapsto [(X_b \cup \{d\},b) \mapsto [(X_c,c) \mapsto [\,]\,]\,]\,]$$

$$\Lsh_{\Phi_2}(M_q,r_p) = [(X_d,d) \mapsto [(X_e,e) \mapsto [\,]\,]\,]$$

$$\begin{aligned} N_{\Phi_2}(r_p,r_q) &= N_{\Phi_2}(r_q,r_p) \\ &= \{(X_a \cup \{e\},a),(X_b \cup \{d\},b),(X_c,c),(X_d,d),(X_e,e)\} \end{aligned}$$

$$\begin{array}{lllllllll}
\looparrowright_{\Phi_2}(r_p,r_q) & = & [(X_a\cup\{e\},a) & \mapsto & [(X_b\cup\{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & &] & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}$$

Let $X'_a = X_a \cup \{e\}$ and $X'_b = X_b \cup \{d\}$. We have

$$\begin{array}{l}
\looparrowright_{\Phi_2}(r_q,r_p) \quad = \\
\begin{array}{lllllllllll}
[(X_d,d) & \mapsto & [(X'_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,] & & & & \\
 & & & &], & & & & & & \\
 & & (X_e,e) & \mapsto & [(X'_a,a) & \mapsto & [(X'_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & & \\
] & & & & & & & & & &
\end{array}
\end{array}$$

$$\begin{array}{l}
r_p \parallel_{\Phi_2} r_q \quad = \\
\begin{array}{lllllllllll}
([(X_d,d) & \mapsto & [(X'_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,] & & & & \\
 & & & &], & & & & & & \\
 & & (X_e,e) & \mapsto & [(X'_a,a) & \mapsto & [(X'_b,b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & & \\
],\lambda_p\cup\lambda_q) & & & & & & & & & &
\end{array}
\end{array}$$

$$r_p \parallel_{\Phi_3} r_q \quad = \quad (\looparrowright_{\Phi_3}(M_p,M_q) \bigcirc \looparrowright_{\Phi_3}(M_q,M_p),\lambda_p\cup\lambda_q)$$

$$\begin{array}{lllllllll}
\Lsh_{\Phi_3}(M_p,r_q) & = & [(X_a,a) & \mapsto & [(X_b\cup\{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & &] & & \\
 & & & &] & & & & \\
 & &] & & & & & & \\
\Lsh_{\Phi_3}(M_q,r_p) & = & [(X_d,d) & \mapsto & [(X_e\cup\{a\},e) & \mapsto & [\,] & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}$$

$$\begin{array}{lll}
N_{\Phi_3}(r_p,r_q) & = & N_{\Phi_3}(r_q,r_p) \\
 & = & \{(X_a,a),(X_b\cup\{d\},b),(X_c,c),(X_d,d),(X_e\cup\{a\},e)\}
\end{array}$$

$$\left(\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \end{tikzcd} , \emptyset \right) \quad \| \quad \left(\begin{tikzcd} d \arrow[r] & e \end{tikzcd} , \emptyset \right) =$$

$$\left\{ \left(\begin{tikzcd} a \arrow[r] \arrow[dr] & b \arrow[r] \arrow[dl] & c \\ d \arrow[r] & e & \end{tikzcd} , \emptyset \right), \left(\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \\ d \arrow[r] \arrow[ur] & e \arrow[ul] & \end{tikzcd} , \emptyset \right), \left(\begin{tikzcd} a \arrow[r] \arrow[dr] & b \arrow[r] & c \\ d \arrow[r] \arrow[ur] & e & \end{tikzcd} , \emptyset \right) \right\}$$

Figure 7.10: A parallel composition of two finite processes

$$\begin{array}{lllllllll}
\looparrowright_{\Phi_3}(r_p,r_q) & = & [(X_a,a) & \mapsto & [(X_b \cup \{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & &], & & \\
 & & & & (X_e \cup \{a\},e) & \mapsto & [\,] & & \\
 & & & &] & & & & \\
 & &] & & & & & & \\
\looparrowright_{\Phi_3}(r_q,r_p) & = & [(X_d,d) & \mapsto & [(X_b \cup \{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & & & &], & & \\
 & & & & (X_e \cup \{a\},e) & \mapsto & [\,] & & \\
 & & & &] & & & & \\
 & &] & & & & & &
\end{array}$$

$$\begin{array}{lllllll}
r_p \parallel_{\Phi_3} r_q & = & & & & & \\
([(X_a,a) & \mapsto & [(X_b \cup \{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (X_e \cup \{a\},e) & \mapsto & [\,] & & \\
 & &], & & & & \\
\;(X_d,d) & \mapsto & [(X_b \cup \{d\},b) & \mapsto & [(X_c,c) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (X_e \cup \{a\},e) & \mapsto & [\,] & & \\
 & &] & & & & \\
],\lambda_p \cup \lambda_q) & & & & & &
\end{array}$$

Example 2: The second example shows a case where we need to compute the maximal prefixes in the composed processes in order to do the composition.

$$\begin{array}{lllllllllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b\},c_1) & \mapsto & [(\{c_1\},c_2) & \mapsto & [(\{c_2\},c_3) \mapsto \ldots \\
 & & & & & & & & & &] \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & &
\end{array}$$

$$\begin{array}{lll}
\lambda_p & = & \{(a,a),(b,b),(c_1,c),(c_2,c),\ldots\} \\
R_p & = & \hat{c}
\end{array}$$

$$\begin{array}{lcl} M_q & = & [(\emptyset,d) \mapsto [(\{d\},e) \mapsto [(\{e\},f_1) \mapsto [(\{f_1\},f_2) \mapsto [(\{f_2\},f_3) \mapsto \ldots \\ & &] \\ & &] \\ & &] \\ & &] \\ & &] \\ \lambda_q & = & \{(d,d),(e,e),(f_1,f),(f_2,f),\ldots\} \\ R_q & = & \hat{f} \end{array}$$

We suppose that we only have $b\ D\ d$ and $c\ D\ f$.

$$\begin{array}{lcl} M_{\mu^p_{R_q}} & = & [(\emptyset,a) \mapsto [(\{a\},b) \mapsto [\,] \\ & &] \\ & &] \\ \lambda_{\mu^p_{R_q}} & = & \{(a,a),(b,b)\} \\ R_{\mu^p_{R_q}} & = & \hat{c} \\ \mu^q_{R_p} & = & [(\emptyset,d) \mapsto [(\{d\},e) \mapsto [\,] \\ & &] \\ & &] \\ \lambda_{\mu^q_{R_p}} & = & \{(d,d),(e,e)\} \\ R_{\mu^p_{R_q}} & = & \hat{f} \end{array}$$

For this parallel composition, there are two possible dependence relations: Φ_1 and Φ_2

$$\Phi_1 = \{(a,b),(b,d),(d,e)\}$$

$$r_{\mu^p_{R_q}} \|_{\Phi_1} r^q_{\mu_{R_p}} =$$

$$\begin{array}{l} ([(\emptyset,a) \mapsto [(\{a\},b) \mapsto [(\{b\},d) \mapsto [(\{d\},e) \mapsto [\,] \\ ] \\ ] \\ ] \\ \lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}}) \end{array}$$

$$R^p_q = \hat{c} \cup \hat{f}$$

$$\Phi_2 = \{(a,b),(d,b),(d,e)\}$$

$$r_{\mu^p_{R_q}} \|_{\Phi_2} r^q_{\mu_{R_p}} =$$

$$\left(a \to b \to c_1 \to c_2 \cdots , \quad \hat{c} \right) \quad \| \quad \left(d \to e \to f_1 \to f_2 \cdots , \quad \hat{f} \right) =$$

$$\uparrow \left\{ \left(\begin{tikzcd} a \arrow[r] & b \arrow[dl] \\ d \arrow[r] & e \end{tikzcd} , \quad \hat{c} \cup \hat{f} \right), \left(\begin{tikzcd} a \arrow[r] & b \\ d \arrow[r] \arrow[ur] & e \end{tikzcd} , \quad \hat{c} \cup \hat{f} \right) \right\}$$

Figure 7.11: A parallel composition of two infinite processes

$$\begin{array}{lllll}
([(\emptyset,a) & \mapsto & [(\{a,d\},b) & \mapsto & [\,] \\
 & &], & & \\
(\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto & [\,], \\
 & & (\{a,d\},b) & \mapsto & [\,] \\
 & &] & & \\
\multicolumn{5}{l}{\lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}})} \\
R^p_q & = & \hat{c} \cup \hat{f} & &
\end{array}$$

A graphical representation of the second example is outlined in Figure 7.11.

Example 3: The third example shows a case where the continuation resources of the computed maximal prefixes are claimed by the process, which is the result of the composition.

$$\begin{array}{lllllllll}
M_p & = & [(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [(\{b\},c) & \mapsto & [\,] \\
 & & & & & &] & & \\
 & & & &], & & & & \\
 & & (\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto & [\,] & & \\
 & & & &] & & & & \\
 & &] & & & & & & \\
\lambda_p & = & \multicolumn{7}{l}{\{(a,a),(b,b),(c,c),(d,d),(e,e)\}} \\
R_p & = & R_1 & & & & & &
\end{array}$$

$$\begin{array}{lllllllllll}
M_q & = & [(\emptyset,f) & \mapsto & [(\{f\},g_1) & \mapsto & [(\{g_1\},g_2) & \mapsto & [(\{g_2\},g_3) & \mapsto & \ldots \\
 & & & & & & & &] & & \\
 & & & & & &] & & & & \\
 & & & &] & & & & & & \\
 & &] & & & & & & & & \\
\lambda_q & = & \multicolumn{9}{l}{\{(f,f),(g_1,g),(g_2,g),\ldots\}} \\
R_q & = & R_2 & & & & & & & &
\end{array}$$

We suppose that we only have $e\ D\ f$ and that $R_1 \cap \hat{g} \neq \emptyset$ and $R_2 \cap \hat{c} \neq \emptyset$.

$$\begin{array}{lcllll}
\mu^p_{R_2} & = & ((([(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto \; [\,] \\
 & & & &], & \\
 & & (\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto \; [\,] \\
 & & & &] & \\
 & & \lambda_{\mu^p_{R_2}}),R_1 \cup \hat{c}) & & & \\
\lambda_{\mu^p_{R_2}} & = & \{(a,a),(b,b),(d,d),(e,e)\} & & & \\
\mu^q_{R_1} & = & ((([(\emptyset,f) \mapsto [\,]],\lambda_{\mu^q_{R_1}}),R_2 \cup \hat{g}) & & & \\
\lambda_{\mu^q_{R_1}} & = & \{(f,f)\} & & &
\end{array}$$

For this parallel composition, there are two possible dependence relations: Φ_1 and Φ_2

$$\Phi_1 = \{(a,b),(d,e),(e,f)\}$$

$$\begin{array}{lcllllll}
r_{\mu^p_{R_2}} \parallel_{\Phi_1} r_{\mu^q_{R_1}} & = & ([(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (\emptyset,d) & \mapsto & [(\{d\},e) & \mapsto & [(\{e\},f) \mapsto [\,] \\
 & & & & & &] \\
 & & & &] & & \\
 & & \lambda_{\mu^p_{R_2}} \cup \lambda_{\mu^q_{R_1}}) & & & &
\end{array}$$

$$\Phi_2 = \{(a,b),(d,e),(f,e)\}$$

$$\begin{array}{lcllllll}
r_{\mu^p_{R_2}} \parallel_{\Phi_2} r_{\mu^q_{R_1}} & = & ([(\emptyset,a) & \mapsto & [(\{a\},b) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (\emptyset,d) & \mapsto & [(\{d,f\},e) & \mapsto & [\,] \\
 & & & &], & & \\
 & & (\emptyset,f) & \mapsto & [(\{d,f\},e) & \mapsto & [\,] \\
 & & & &] & & \\
 & & \lambda_{\mu^p_{R_2}} \cup \lambda_{\mu^q_{R_1}}) & & & &
\end{array}$$

$$R^p_q = R_1 \cup R_2 \cup \hat{c} \cup \hat{g}$$

Finally, we get

$$p \parallel q \;=\; \uparrow\{(r_{\mu^p_{R_2}} \parallel_{\Phi_1} r_{\mu^q_{R_1}},R_1 \cup R_2 \cup \hat{c} \cup \hat{g}),\ (r_{\mu^p_{R_2}} \parallel_{\Phi_2} r_{\mu^q_{R_1}},R_1 \cup R_2 \cup \hat{c} \cup \hat{g})\}$$

The third example is outlined in Figure 7.12.

7.5.7 Recursion

In what follows, we establish a proposition that is useful to define the recursion semantics.

$$\left(\begin{tikzcd} a \arrow[r] & b \arrow[r] & c \\ d \arrow[r] & e & \end{tikzcd} , R_1\right) \parallel \left(\begin{tikzcd} f \arrow[r] & g_1 \arrow[r] & g_2 \cdots \end{tikzcd} , R_2\right) =$$

$$\uparrow\left\{\left(\begin{tikzcd} a \arrow[r] & b & f \\ d \arrow[r] & e \arrow[ur] & \end{tikzcd} , R_1 \cup R_2 \cup \hat{c} \cup \hat{g}\right), \left(\begin{tikzcd} a \arrow[r] & b & f \arrow[dl] \\ d \arrow[r] & e & \end{tikzcd} , R_1 \cup R_2 \cup \hat{c} \cup \hat{g}\right)\right\}$$

Figure 7.12: A parallel composition of two non-terminated processes

Proposition 7.5.1 *Let Ord be the class of ordinals, P be a local cpo and $E \subseteq P$ be a cpo. Suppose that $f : P \to P$ is monotone and satisfies the property that there is some monotone mapping $F : E \to E$ with $\forall y \in E.f(y) \sqsubseteq F(y)$, i.e. F dominates f with respect to the order $\sqsubseteq$. If F has a postfixed point x in E, i.e. $x \sqsubseteq F(x)$ and x is a postfixed point of f in P, then the directed set $\{f^\alpha(x) \mid \alpha \in Ord\}$ has a least upper bound in P.*

Proof Let x be a postfixed point of F and f. Since E is a cpo, the directed set $\{F^\alpha(x) \mid \alpha \in Ord\}$ has a least upper bound. Hence, $\bigsqcup_{\alpha\in Ord} F^\alpha(x)$ exists. Using the fact that F dominates f, we deduce easily that the set $\{f^\alpha(x) \mid \alpha \in Ord\}$ is bounded by $\bigsqcup_{\alpha\in Ord} F^\alpha(x)$. Since $\mathbb{D}$ is a local cpo, every directed set, which is bounded, i.e. has an upper bound, has a least upper bound. Hence, we deduce that $\bigsqcup\{f^\alpha(x) \mid \alpha \in Ord\}$ exists. ∎

Let $f : \mathbb{D} \to \mathbb{D}$ defined by

$$f(Y) \quad = \quad [\![P]\!](\theta[X \mapsto Y]) \cap \uparrow (([\,], \emptyset), \mathcal{R})$$

It is worth to mention that the intersection with $\uparrow (([\,], \emptyset), \mathcal{R})$ allows f^α, for $\alpha \in Ord$, to satisfy the second healthiness condition. Moreover, $\theta[X \mapsto Y]$ means that the environment θ is updated to have the value Y at X.

Let $Z = \uparrow (([\,], \emptyset), \emptyset)$ and $X_0 = \uparrow (([\,], \emptyset), res(f(Z)))$. The recursion semantics is defined as follows:

$$[\![rec\ X.\ P]\!](\theta) \quad = \quad \bigsqcup_{\alpha\in Ord} f^\alpha(X_0)$$

As defined, the recursion semantics does not use the least element, $\uparrow (([\,], \emptyset), \mathcal{R})$, as a starting point. The reason underlying such a decision is that the resources specified for this element are more than what it is actually required to do the recursion. Composing this element with other processes leads to a non desired result. For instance, in this case, a parallel composition of a recursive process that is executing infinite events, which are instances of an action a, with another one executing infinite events, which are instances of an action b, leads to a deadlock. Accordingly, we adopt a semantics

$$\uparrow \left(a_1 \rightarrow a_2 \rightarrow a_3 \quad \cdots , \quad \hat{a} \right)$$

Figure 7.13: The semantics of the process P

for the recursion where the resources of the starting point depend on the definition of the recursive process.

Note that the proofs related to the fixed point existence are provided after those dealing with the monotonicity of our semantic functions. The strategy consists of finding for each semantic function $\tilde{\omega}^+$ a dominating function F that satisfies the constraints of proposition 7.5.1. We also prove later that the output of each semantic function satisfies the two healthiness conditions.

Examples:

The following examples illustrate the recursion semantics in our model.

In the first example, we provide the semantics of a simple recursive process. Note that a^k denotes the dependence map having k nodes and where $(\emptyset, a_1)$ is the unique initial of this map and $\forall 1 < j \leq k$. a_j is preceded only by a_{j-1} and a^ω denotes an infinite sequence of a.

$$\begin{array}{lcl}
P & = & rec\ X.\ \mathrm{a}; X \\
X_0 & = & \uparrow(a, \hat{a}) \\
X_1 & = & \uparrow(a^2, \hat{a}) \\
\ldots & & \\
X_\omega & = & \uparrow(a^\omega, \hat{a}) \\
X_\beta & = & \sqcup_{\alpha<\beta} X_\alpha = \uparrow(a^\omega, \hat{a}) \text{ for } \beta \text{ a limit ordinal.}
\end{array}$$

The semantics of the previous process is outlined in Figure 7.13.

In what follows, we suppose that a and b are independent.

$$Q \quad = \quad (rec\ X.\ \mathrm{a}; X) \ \| \ (rec\ X.\ b; X)$$

Figure 7.14 provides the semantics of the process Q.

Hereafter, we also suppose that a and b are independent.

$$T \quad = \quad rec\ X.\ ((\mathrm{a} \parallel b); e); X$$

Figure 7.15 outlines the semantics of the process T.

$$\uparrow\left(\begin{tikzcd} a_1 \arrow[r] & a_2 \arrow[r] & a_3 & \cdots \\ b_1 \arrow[r] & b_2 \arrow[r] & b_3 & \cdots \end{tikzcd}, \quad \hat{a}\cup\hat{b}\right)$$

Figure 7.14: The semantics of the process Q

$$\uparrow\left(\begin{tikzcd} a_1 \arrow[dr] & & a_2 \arrow[dr] & \\ & e_1 \arrow[ur] \arrow[dr] & & e_2 \cdots \\ b_1 \arrow[ur] & & b_2 \arrow[ur] & \end{tikzcd}, \quad \hat{a}\cup\hat{b}\cup\hat{e}\right)$$

Figure 7.15: The semantics of the process T

7.6 Details about the Construction of the Space $\mathbb{M}$

In what follows, we give an overview of the transfinite recursive space construction technique, proposed by Di Gianantonio et al. [53] and which we use to construct the maps space $\mathbb{M}$.

Let D be a recursive space to build and C the category where objects are cpos and morphisms are w_1-continuous functions. Let $F : C \to C$ be a functor. The diagram outlined in Figure 7.16 defines a condition that should be satisfied by F. It guaranties that for each chain $\langle D_\alpha, e_{\alpha,\beta}\rangle_{\alpha<\beta<\lambda}$, where λ is a countable limit ordinal, $D_{\alpha+1} = F(D_\alpha)$ and $e_{\alpha+1,\beta+1} = F(e_{\alpha,\beta})$ there exists a cone $\mu : \langle D_\alpha, e_{\alpha,\beta}\rangle_{\alpha<\beta<\lambda} \to D_\lambda$ and a morphism e_λ such that $\forall\alpha < \lambda.\ e_\lambda \circ \mu_{\alpha+1} = F(\mu_\alpha)$. The space D is built through the construction of a chain of spaces $\langle D_0, F(D_0), F^2(D_0), \ldots\rangle$. Each space D_α is related to $D_{\alpha+1}$ by a morphism $e_{\alpha,\alpha+1}$, which corresponds to e_α in the diagram. More precisely, the construction technique is defined as follows:

- $D_1 = F(D_0)$ and $e_{0,1} = e_0$
- For $\beta = \beta' + 2$, $\beta' \geq 0$:
  - $D_\beta = F(D_{\beta'+1})$
  - $e_{\beta'+1,\beta} = F(e_{\beta',\beta'+1})$
  - $e_{\alpha,\beta} = e_{\beta'+1,\beta} \circ e_{\alpha,\beta'+1}\ \forall\alpha \leq \beta$

$$\begin{tikzcd}
D_0 \arrow[r, "e_0"] \arrow[dr, "\mu_0"] & F(D_0) \arrow[r, "F(e_0)"] \arrow[d, "\mu_1"] \arrow[dr, "F(\mu_0)"'] & F^2(D_0) \arrow[r] \arrow[dl, "\mu_2"] \arrow[d, "F(\mu_1)\ldots"] & \cdots \\
& D_\lambda \arrow[r, "e_\lambda"] & F(D_\lambda) &
\end{tikzcd}$$

Figure 7.16: Cone Existence Diagram

- For β a limit ordinal, $\langle e_{\alpha,\beta}\rangle_\alpha : \langle D_\alpha, e_{\alpha,\gamma}\rangle_{\alpha<\gamma<\beta} \to D_\beta$ is the cone whose existence is ensured by the diagram,
- For $\beta = \lambda + 1$ with λ a limit ordinal,
  - $D_\beta = F(D_\lambda)$
  - $e_{\lambda,\lambda+1}$ is the morphism whose existence is guaranteed by the diagram.
  - $e_{\alpha,\lambda+1} = e_{\lambda,\lambda+1} \circ e_{\alpha,\lambda}\ \forall\alpha < \lambda$

In what follows, we apply the already described technique to construct the space $\mathbb{M}$. Let $\to_{\omega_1}$ be the constructor of infinite maps in which an element can be associated with a transfinite number of elements. The space $\mathbb{M}$ as follows:

$$\mathbb{M} \quad = \quad \mathcal{P}_f(V) \times V \to_{\omega_1} \mathbb{M}$$

The space $\mathbb{M}$ is a solution to the equation $X = F(X)$ where F is a functor such that $\forall X \in C.\ F(X) = \mathcal{P}_f(V) \times V \to_{\omega_1} X$. This constructor is ω_1-continuous since it is the composition of the functors $\times$ and $\to_{\omega_1}$, which are ω_1-continuous as proved in [53]. Moreover, the composition of these functors satisfies the condition defined by the diagram of figure 7.16 as stated in [53].

Let $\mathbb{M}_0 = \{\bot\} = \{[\,]\}$ and $\rhd$ be an ordering over $\mathbb{M}$ defined as follows:

Let M, $M' \in \mathbb{M}$. We have

1. $[\,] \ \rhd \ M$
2. $M \rhd M' \Leftrightarrow dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ M(a) \rhd M'(a)$

The construction of $\mathbb{M}$ is as follows:

$$\begin{array}{lcl} e_{0,1} & : & \mathbb{M}_0 \to \mathbb{M}_1 \\ e_{0,1} & = & \lambda x.\{\bot_{\mathbb{M}_1}\}, \text{ where } \mathbb{M}_1 = F(\mathbb{M}_0) \text{ and } \bot_{\mathbb{M}_1} = [\,] \end{array}$$

For $\beta = \beta' + 2$, $\beta' \geq 0$:

- $\mathbb{M}_\beta = F(\mathbb{M}_{\beta'+1})$
- $e_{\beta'+1,\beta} = F(e_{\beta',\beta'+1})$
- $e_{\alpha,\beta} = e_{\beta'+1,\beta} \circ e_{\alpha,\beta'+1}\ \forall \alpha \leq \beta$

For β a limit ordinal, $\langle e_{\alpha,\beta}\rangle_\alpha : \langle \mathbb{M}_\alpha, e_{\alpha,\gamma}\rangle_{\alpha<\gamma<\beta} \to \mathbb{M}_\beta$ is the cone whose existence is assured by the diagram.

For $\beta = \lambda + 1$ with λ a limit ordinal:

- $\mathbb{M}_\beta = F(\mathbb{M}_\lambda)$
- $e_{\lambda,\lambda+1}$ is the morphism whose existence is guaranteed by the diagram.
- $e_{\alpha,\lambda+1} = e_{\lambda,\lambda+1} \circ e_{\alpha,\lambda}\ \forall \alpha < \lambda$

Hence, we proved the existence of the space $\mathbb{M}$. It is worth to mention that the construction is limited to the subcategory where morphisms are embeddings (see [53] for more details).

7.7 Algebraic Properties

In order to give a meaning to recursion, we have to establish some results about our process space. Before this, we motivate for the need of a co-induction principle to establish such results.

7.7.1 Definition of a Co-induction Principle over the Space $\mathbb{M}$

The space $\mathbb{M}$ contains infinite maps. To compare two infinite maps using the order $\rhd$, we need to define a co-induction principle over $\mathbb{M}$. Before discussing the need for co-induction, we recall the following theorem.

Theorem 7.7.1 (Knaster-Tarski) *Let $(A,\sqsubseteq)$ be a cpo and $f : A \to A$ a monotone function then $\sqcap\{x \in A \mid f(x) \sqsubseteq x\}$ is the least fixed point of f and $\sqcup\{x \in A \mid x \sqsubseteq f(x)\}$ is the greatest fixed point of f.*

Let $\mathcal{U}$ be the set of all (M, M') such that M, $M' \in \mathbb{M}$. It is clear that $(\mathcal{P}(\mathcal{U}), \subseteq)$ is a cpo. In fact, the least upper bound of a directed set in $\mathcal{P}(\mathcal{U})$ is the union of the elements of this set.

Let $\mathcal{F}$ be a function defined as follows:

$$\begin{array}{lcl}\mathcal{F} & : & \mathcal{P}(\mathcal{U}) \rightarrow \mathcal{P}(\mathcal{U}) \\ \mathcal{F}(Q) & = & \{(M, M') \mid dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ (M(a), M'(a)) \in Q\}\end{array}$$

The function $\mathcal{F}$ has a greatest fixed point (gfp). To prove that this gfp exists, it is sufficient to prove that $\mathcal{F}$ is monotone since $(\mathcal{P}(\mathcal{U}), \subseteq)$ is a cpo.

Proposition 7.7.2 *The function $\mathcal{F}$ has a greatest fixed point.*

Proof Let Q,$Q' \in \mathcal{P}(\mathcal{U})$ such that $Q \subseteq Q'$, we claim that $\mathcal{F}(Q) \subseteq \mathcal{F}(Q')$. In what follows, we prove this claim.

Let $q = (M, M') \in \mathcal{F}(Q)$. We have

$dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ (M(a), M'(a)) \in Q$

This means that

$dom(M) \subseteq dom(M') \wedge \forall a \in dom(M).\ (M(a), M'(a)) \in Q'$ since $Q \subseteq Q'$

This proves that $q \in \mathcal{F}(Q')$ and finally that $\mathcal{F}(Q) \subseteq \mathcal{F}(Q')$.

By theorem 7.7.1, we conclude that $\nu\mathcal{F} = \bigcup\{Q \in \mathcal{P}(\mathcal{U}) \mid Q \subseteq \mathcal{F}(Q)\}$. Note that in our proofs, $\nu\mathcal{F}$ will be denoted by $\overline{\mathcal{F}}$.

It is important to note that $\overline{\mathcal{F}}$ captures all the sets of pairs composed by maps that are related by the prefixing relation $\rhd$. Hereafter, we present an example showing that the least fixed point of the function $\mathcal{F}$ cannot capture all the related elements.

Let $Q = \{([\,], M) \mid M \in \mathbb{M}\}$. Let $q = ([(\emptyset, a) \mapsto [\,]], M')$ such that $(\emptyset, a) \in dom(M')$. We have $q \in \mathcal{F}(Q)$ but $q \notin Q$. Hence, it is clear that $\mathcal{F}(Q) \nsubseteq Q$ and consequently $q \notin \mu\mathcal{F}$.

Note that the co-induction technique, defined above, can be also applied over the space $\mathbb{T}$ since it is a subspace of the space $\mathbb{M}$.

7.7.2 Algebraic Properties: Claims and Proofs

In this section, we provide the details about the proofs related to the algebraicity of the deterministic process space $\mathbb{C}$.

($\mathbb{C}$,$\sqsubseteq_{\mathbb{C}}$) is a consistently complete partial order (ccpo)

Proof First, we claim that $(\mathbb{R}, \preceq)$ is a ccpo. Let X be a consistent set that is subset of $\mathbb{R}$. In what follows, we prove that X has a least upper bound, which is

$u = (\bigvee_{x \in X} M_x, \bigcup_{x \in X} \lambda_x)$ where $\vee$ is defined as follows:

$$\begin{array}{rcl} M \vee M' & = & [a \mapsto M(a) \vee M'(a) \mid a \in dom(M) \cap dom(M')] \dagger \\ & & [a \mapsto M(a) \mid a \in dom(M) \setminus dom(M')] \dagger \\ & & [a \mapsto M'(a) \mid a \in dom(M') \setminus dom(M)] \end{array}$$

First, we have to prove that u is an upper bound of X.

Let $t \in X$. We have to prove that

$$\begin{cases} M_t \rhd \bigvee_{x \in X} M_x \\ \lambda_t \subseteq \bigcup_{x \in X} \lambda_x \end{cases}$$

The proof of the last constraint is obvious.

For the first constraint, the proof is done using the coinductive principle that we have already defined.

Let $Q \in \mathcal{P}(\mathcal{U})$ such that

$$\begin{array}{rcl} Q & = & \{(M_z, \bigvee_{x \in Z} M_x) \mid Z \in \mathcal{P}((\mathbb{M} \times (V \overset{\sim}{\longrightarrow} \Sigma)) \times \mathcal{P}(\mathcal{R})) \wedge z \in Z\} \cup \\ & & \{(M, M) \in \mathbb{M}^2\} \end{array}$$

Let $(M_z, \bigvee_{x \in Z} M_x) \in Q$. We have to prove that $(M_z, \bigvee_{x \in Z} M_x) \in \mathcal{F}(Q)$.

We have $dom(M_z) \subseteq dom(\bigvee_{x \in Z} M_x)$ since we have

$$dom(\bigvee_{x \in Z} M_x) = \bigcup_{x \in Z} dom(M_x)$$

Moreover, let $a \in dom(M_z)$ and $J_a = \{x \in Z \mid a \in dom(M_x) \cap dom(M_z)\}$. If $J_a = \{z\}$, then we have

$$(\bigvee_{x \in Z} M_x)(a) = M_z(a)$$

In this case, it is clear that $(M_z(a), M_z(a)) \in Q$.

Otherwise, from the definition of the operator $\vee$, it follows that

$$(\bigvee_{x \in Z} M_x)(a) = \bigvee_{x \in J_a} M_x(a)$$

Since we can find J'_a such that $\bigvee_{x \in J_a} M_x(a) = \bigvee_{x' \in J'_a} M_{x'}$ and $M_{x'} = M_x(a)$, we deduce that $(M_z(a), \bigvee_{x \in J_a} M_x(a)) \in Q$.

Now, let $(M, M) \in Q$. Since $\rhd$ is reflexive, we have $M \rhd M$ and we deduce that $(M, M) \in \mathcal{F}(Q)$. This concludes the proof $Q \subseteq \mathcal{F}(Q)$.

Now, since $t \in X$, it is clear that $(M_t, \bigvee_{x \in X} M_x) \in Q$ by taking $Z = X$. Hence, we deduce that we have particularly $(M_t, \bigvee_{x \in X} M_x) \in \mathcal{F}(Q)$. This means that $M_t \rhd \bigvee_{x \in X} M_x$ and proves that u is an upper bound of X.

After this, we have to prove that u is the least one of the upper bounds. Let y be an upper bound of X. Let us prove that $u \preceq y$.

First, we have $\bigcup_{x\in X} \lambda_x \subseteq \lambda_y$ since $\forall x \in X.\ \lambda_x \subseteq \lambda_y$. Second, let

$$\begin{array}{rcl} Q & = & \{(\bigvee_{x\in Z} M_x, M) \mid Z \in \mathcal{P}((\mathbb{M} \times (V \xrightarrow{\sim} \Sigma)) \times \mathcal{P}(\mathcal{R})) \wedge \forall x \in Z.\ M_x \rhd M\} \cup \\ & & \{(M, M') \in \mathbb{M}^2 \mid M \rhd M'\} \end{array}$$

Let $(\bigvee_{x\in Z} M_x, M) \in Q$. Let us prove that $(\bigvee_{x\in Z} M_x, M) \in \mathcal{F}(Q)$.

We have $dom(\bigvee_{x\in Z} M_x) = \bigcup_{x\in Z} dom(M_x)$. Since $\forall x \in Z.\ M_x \rhd M$, we have

$\forall x \in Z.\ dom(M_x) \subseteq dom(M)$

This means that $dom(\bigvee_{x\in Z} M_x) \subseteq dom(M)$.

Moreover, let $a \in dom(\bigvee_{x\in Z} M_x)$ then $\exists z \in Z$ such that $a \in dom(M_z)$. Let $J_a = \{x \in Z \mid a \in dom(M_z) \cap dom(M_x)\}$.

If $J_a = \{z\}$, we have

$(\bigvee_{x\in Z} M_x)(a) = M_z(a)$

Hence, it is clear that $(M_z(a), M(a)) \in Q$ since $M_z(a) \rhd M(a)$.

Otherwise, from the definition of the operator $\vee$, it follows that

$(\bigvee_{x\in Z} M_x)(a) = \bigvee_{x\in J_a} M_x(a)$

Since $\forall x \in Z.\ M_x(a) \rhd M(a)$, it is clear that $(\bigvee_{x\in J_a} M_x(a), M(a)) \in Q$.

This means that $(\bigvee_{x\in Z} M_x, M) \in \mathcal{F}(Q)$.

Besides, let $(M, M') \in Q$. Since $M \rhd M'$, we have $(M, M') \in \overline{\mathcal{F}}$.

From this, we deduce that $Q \subseteq \mathcal{F}(Q)$, which means that $(\bigvee_{x\in Z} M_x, M) \in \overline{\mathcal{F}}$.

Now, it is clear that we have particularly $(\bigvee_{x\in X} M_x, M_y) \in Q$ since $\forall x \in X.\ M_x \rhd M_y$.

Hence, we deduce that $\bigvee_{x\in X} M_x \rhd M_y$ and we conclude that u is the least upper bound of X. This proves the claim that $(\mathbb{R}, \preceq)$ is a ccpo.

Now, let X be a consistent set that is subset of $\mathbb{C}$.

We claim that $\sqcup X = ((\bigvee_{x\in X} M_x, \bigcup_{x\in X} \lambda_x), \bigcap_{x\in X} R_x)$.

Let $u = ((\bigvee_{x\in X} M_x, \bigcup_{x\in X} \lambda_x), \bigcap_{x\in X} R_x)$.

First, we have to prove that u is an upper bound of X, then that it is the least one.

Let $t \in X$. We have to prove that

$$\begin{cases} M_t \rhd \bigvee_{x\in X} M_x \\ \lambda_t \subseteq \bigcup_{x\in X} \lambda_x \\ R_t \supseteq \bigcap_{x\in X} R_x \cup res(r_t^{-1} r_u) \end{cases}$$

The first two constraints are verified in the previous proof. Therefore, we only have to prove that $R_t \supseteq \bigcap_{x\in X} R_x \cup res(r_t^{-1} r_u)$. We know that $R_t \supseteq \bigcap_{x\in X} R_x$ and we have to prove that $R_t \supseteq res(r_t^{-1} r_u)$.

If $r_t = r_u$, then the constraint holds since in this case $res(r_t^{-1} r_u) = \emptyset$. Otherwise, let $R \in res(r_t^{-1} r_u)$. We have $r_u = \sqcup_{x\in X} r_x$. Hence, $\exists s \in X$ and $\exists a \in \xi(r_s)$ such that $a \notin \xi(r_t)$ and $R \in res(\lambda_s(a))$. Since X is a consistent set, there exists $z = (r_z, R_z)$ such that $t \sqsubseteq z$ and $s \sqsubseteq z$. Since $r_s \preceq r_z$, we have $\xi(r_s) \subseteq \xi(r_z)$. Hence, $R \in res(r_t^{-1} r_z) \subseteq R_t$. Consequently, we deduce that $R_t \supseteq res(r_t^{-1} r_u)$. This proves that u is an upper bound of X.

Let us prove that it is the least one. Let y be an upper bound of X. This means that for all $x \in X$, we have

$$\begin{cases} M_x \rhd M_y \\ \lambda_x \subseteq \lambda_y \\ R_x \supseteq R_y \cup res(r_x^{-1} r_y) \end{cases}$$

Since r_u is the least upper bound of the set $\{r_x \mid x \in X\}$, we have

(7.6) $\quad r_u \preceq r_y$

Since we have $\forall x \in X.\ r_x \preceq r_u$, we deduce, using equation 7.3, that

$$res(r_x^{-1} r_y) = res(r_x^{-1} r_u) \cup res(r_u^{-1} r_y)$$

This means that $res(r_u^{-1} r_y) \subseteq res(r_x^{-1} r_y)$.

Hence, $\forall x \in X.\ R_y \cup res(r_u^{-1} r_y) \subseteq R_y \cup res(r_x^{-1} r_y) \subseteq R_x$.

Consequently, we have

(7.7) $\quad \bigcap_{x\in X} R_x \supseteq R_y \cup res(r_u^{-1} r_y)$

Using 7.6 and 7.7, we deduce that $u \sqsubseteq y$. Hence, u is the least upper bound of X and this proves the claim that each consistent set X that is subset of $\mathbb{C}$ has a least upper bound.

It is worth to note that we can show easily that $\mathcal{D}(\varphi(M_u)) = \bigcup_{x\in X} \mathcal{D}(\varphi(M_x))$ and that the reflexive transitive closure $\mathcal{D}(\varphi(M_u))$ is a partial order relation using the fact that X is consistent. Moreover, using equation 7.5, it is straightforward to prove that for all $x \in X$, we have $resinf(r_u) \subseteq R_x$. This means that $resinf(r_u) \subseteq \bigcap_{x\in X} R_x$. Hence, u satisfies the two healthiness conditions. ▮

($\mathbb{C}$,$\sqsubseteq_{\mathbb{C}}$) is algebraic. The compact elements of $\mathbb{C}$ are processes having a finite events set

Proof Let D be a directed set that is subset of $\mathbb{C}$. Let $u = \sqcup D$ and x be a finite process such that $x \sqsubseteq u$. This means that

$$\begin{cases} M_x \rhd \bigvee_{y \in D} M_y \\ \lambda_x \subseteq \bigcup_{y \in D} \lambda_x \\ R_x \supseteq \bigcap_{y \in D} R_y \cup res(r_x^{-1} r_u) \end{cases}$$

Since x is finite, r_x is finite and M_x is a finite map, i.e. $\varphi(M_x)$ is finite (recall that $\varphi(M_x)$ denotes the set of elements of M_x). Hence, we can find a finite subset $F = \{y_0, y_1, \ldots, y_n\}$ of D such that $\varphi(M_x) \subseteq \varphi(\bigvee_{y_j \in F} M_{y_j})$. Moreover, it is straightforward to prove that $\lambda_x \subseteq \bigcup_{y_i \in F} \lambda_{y_i}$.

Let $Q = \{(M, M') \mid M \text{ is finite} \wedge \varphi(M) \subseteq \varphi(M')\}$. We can easily prove that $Q \subseteq \mathcal{F}(Q)$. Since $(M_x, \bigvee_{y_j \in F} M_{y_j}) \in Q$, we deduce that $M_x \rhd \bigvee_{y_j \in F} M_{y_j}$.

Now, since D is directed and $F \subseteq D$ is finite, F has an upper bound t in D. This means that $M_x \rhd M_t$ since $\bigvee_{y_j \in F} M_{y_j} \rhd M_t$. Hence, we deduce that

$$\begin{cases} M_x \rhd M_t \\ \lambda_x \subseteq \lambda_t \end{cases}$$

Now, we claim that there exists $x' \in D$ such that $res(r_{x'}) = res(r_{\sqcup D})$ and $R_{x'} = R_{\sqcup D}$. The proof of this claim is inspired by [50].

Let $R \in res(r_{\sqcup D})$. There exists $x_R \in D$ such that $R \in res(r_{x_R})$. Since D is directed, the finite family $\{x_R \mid R \in res(r_{\sqcup D})\}$ has an upper bound $x_1 \in D$ (this family is finite since $res(r_{\sqcup D})$ is finite). Hence, $R \in res(r_{x_1})$ and we deduce that $res(r_{\sqcup D}) \subseteq res(r_{x_1})$. Since $x_1 \sqsubseteq \sqcup D$, we have $res(r_{x_1}) \subseteq res(r_{\sqcup D})$. Hence, we have $res(r_{x_1}) = res(r_{\sqcup D})$.

Let $R \in R_t \setminus R_{\sqcup D}$. There exists $x_R \in D$ such that $R \notin R_{x_R}$. Since D is directed, there exists $x_2 \in D$ such that $x_R \sqsubseteq x_2$ and $t \sqsubseteq x_2$. This means that we have

$$R_{x_2} \subseteq R_t \cap (\bigcap_{R \in R_t \setminus R_{\sqcup D}} R_{x_R}) \subseteq R_{\sqcup D} = \bigcap_{y \in D} R_y$$

Since D is directed, the finite set $\{x_1, x_2\}$ has an upper bound in D. Let x' be this upper bound. Hence, we have $x_1 \sqsubseteq x'$ and $x_2 \sqsubseteq x'$. We also have

$R_{\sqcup D} \subseteq R_{x'} \subseteq R_{x_2} \subseteq R_{\sqcup D}$ and $res(r_{\sqcup D}) = res(r_{x_1}) \subseteq res(r_{x'}) \subseteq res(r_{\sqcup D})$.

Hence, we conclude that $R_{x'} = R_{\sqcup D}$ and $res(r_{x'}) = res(r_{\sqcup D})$

Let $z \in D$ be an upper bound of t and x'. We have $t \sqsubseteq z$.

Hence, we have

(7.8) $r_x \preceq r_z$ since $r_x \preceq r_t$

We can easily prove that $R_z = R_{\sqcup D}$ since $R_{\sqcup D} \subseteq R_z$, $R_{x'} = R_{\sqcup D}$ and $R_z \subseteq R_{x'}$. Moreover, we have

(7.9) $R_z \cup res(r_x^{-1} r_z) \subseteq R_{\sqcup D} \cup res(r_x^{-1} r_{\sqcup D}) \subseteq R_x$ since $R_z = R_{\sqcup D}$ and $x \sqsubseteq \sqcup D$

From (7.8) and (7.9), it follows that $x \sqsubseteq z$ and we conclude that x is compact.

Now, let $x \in \mathbb{C}$ and $K(x) = \{(r_s, res(s^{-1}x))$ such that s is finite and $r_s \preceq r_x\}$. We claim that $x = \bigsqcup K(x)$.

First, we have to prove that $K(x)$ is a directed set in $\mathbb{C}$. Let F a finite subset of $K(x)$, y_1 and $y_2 \in F$ such that $y_1 = (r_{y_1}, res(y_1^{-1}x))$ and $y_2 = (r_{y_2}, res(y_2^{-1}x))$. We have $r_{y_1} \preceq r_x$ and $r_{y_2} \preceq r_x$. Hence, since $(\mathbb{R}, \preceq)$ is a ccpo, the consistent set $\{r_{y_1}, r_{y_2}\}$ has a least upper bound $r_z \preceq r_x$. Moreover, we can find z_1 and z_2 such that $z_1 = (r_z, \emptyset)$ and $z_2 = (r_{z_1}, res(z_1^{-1}x))$. It is clear that $z_2 \in K(x)$. Moreover, knowing that $res(r_{y_1}^{-1} r_x) = res(r_{y_1}^{-1} r_{z_1}) \cup res(r_{z_1}^{-1} r_x)$, we get

$$res(y_1^{-1}x) \supseteq res(z_1^{-1}x) \cup res(y_1^{-1}z_1) \supseteq res(z_1^{-1}x) \cup res(r_{y_1}^{-1} r_{z_1})$$

We also have $res(y_2^{-1}x) \supseteq res(z_1^{-1}x) \cup res(y_2^{-1}z_1) \supseteq res(z_1^{-1}x) \cup res(r_{y_2}^{-1} r_{z_1})$. Hence, z_2 is an upper bound of y_1 and y_2. This can be generalized on the elements of F. Hence, F has an upper bound in $K(x)$ and consequently $K(x)$ is a directed set.

It is easy to prove that $r_x = \bigsqcup_{r_s \preceq r_x} r_s$. In fact, it is clear that $\xi(r_x) = \bigcup_{r_s \preceq r_x} \xi(r_s)$. Hence, we deduce that $res((\bigsqcup_{r_s \preceq r_x} r_s)^{-1} r_x) = \emptyset$. Now, since $\bigsqcup_{r_s \preceq r_x} r_s \preceq r_x$, by equation 7.2, we deduce that $\bigsqcup_{r_s \preceq r_x} r_s = r_x$.

Moreover, we have

$$\begin{array}{rcl} \bigcap_{r_s \preceq r_x} res(s^{-1}x) & = & \bigcap_{r_s \preceq r_x} (res(r_s^{-1} r_x) \cup R_x) \\ & = & \bigcap_{r_s \preceq r_x} res(r_s^{-1} r_x) \cup R_x \end{array}$$

Since for all $r_s \preceq r_x$, r_s is finite, we have

$$\bigcap_{r_s \preceq r_x} res(r_s^{-1} r_x) = \bigcap \{res({r_s}^{-1} r_x) \mid r_s \preceq r_x \wedge r_s \text{ finite}\} = resinf(r_x)$$

Hence, $\bigcap_{r_s \preceq r_x} res(s^{-1}x) = resinf(r_x) \cup R_x$.

We also know from the second healthiness condition that $resinf(r_x) \subseteq R_x$.

Hence, we deduce that $\bigcap_{r_s \preceq r_x} res(s^{-1}x) = R_x$.

This proves the claim that $\forall x \in \mathbb{C}.\ x = \bigsqcup K(x)$.

We have to prove now that the compact elements are exactly the finite processes. Let x be a compact element in $\mathbb{C}$ such that x is not a finite process. This means that $\xi(r_x)$ is infinite. Since we have $x = \bigsqcup K(x)$, which means that $x \sqsubseteq \bigsqcup K(x)$ and since x is supposed to be compact and $K(x)$ is directed, $\exists y \in K(x)$ such that $x \sqsubseteq y$. Hence, $\exists r_y$ finite such that $r_x \preceq r_y$. This means that $\xi(r_x) \subseteq \xi(r_y)$. Knowing that $\xi(r_y)$ is finite, we get a contradiction with the hypothesis: $\xi(r_x)$ is infinite. Hence, the compact elements are exactly the finite processes. ∎

7.8 Monotonicity Proofs

In this section we provide monotonicity proofs of our semantic functions. These proofs are needed to establish the semantics of recursion.

The proofs of monotonicity are based on the following idea: let $\tilde{\omega}^+$ be a semantic function operating on non-deterministic processes. Let P,Q,P',$Q' \in \mathbb{D}$ such that $P \sqsubseteq P'$ and $Q \sqsubseteq Q'$. We have to prove that $\tilde{\omega}^+(P,Q) \sqsubseteq \tilde{\omega}^+(P',Q')$, i.e. $\forall t' \in \tilde{\omega}^+(P',Q')$. $\exists t \in \tilde{\omega}^+(P,Q)$ such that $t \sqsubseteq t'$. Let $t' \in \tilde{\omega}^+(P',Q')$. $\exists p' \in P'$ and $q' \in Q'$ such that $t' = \tilde{\omega}(p',q')$. Since $P \sqsubseteq P'$, $\exists p \in P$ such that $p \sqsubseteq p'$. We also have $Q \sqsubseteq Q'$, thus $\exists q \in Q$ such that $q \sqsubseteq q'$. Let $t = \tilde{\omega}(p,q)$; the monotonicity proof is reduced to the proof that $t \sqsubseteq t'$, i.e. that $\tilde{\omega}$ is monotone. This strategy is followed for the operators ; and $\circ$. It is also followed for the unary operator $\setminus$ by reasoning on deterministic processes.

In what follows, we need to establish two useful lemmas. These lemmas will be used in proving the monotonicity of some of our semantic functions.

Lemma 7.8.1 *Let M_1, $M_2 \in \mathbb{M}$ such that $M_1 \rhd M_2$ and $M_1' \rhd M_2'$.*

if $dom(M_1) \cap dom(M_2') = \emptyset$ then $M_1 \dagger M_1' \rhd M_2 \dagger M_2'$

Proof Let us prove that $(M_1 \dagger M_1', M_2 \dagger M_2') \in \overline{\mathcal{F}}$. Let $Q \in \mathcal{P}(\mathcal{U})$ such that

$$\begin{array}{rcl} Q & = & \{(M_1 \dagger M_1', M_2 \dagger M_2') \mid M_1 \rhd M_2 \wedge M_1' \rhd M_2'\} \cup \\ & & \{(M_1(a), M_2(a)) \mid a \in dom(M_1) \wedge M_1 \rhd M_2\} \cup \\ & & \{(M_1'(a), M_2'(a)) \mid a \in dom(M_1') \wedge M_1' \rhd M_2'\} \end{array}$$

We have

$$\begin{array}{rcl} dom(M_1 \dagger M_1') & = & dom(M_1) \cup dom(M_1') \\ dom(M_2 \dagger M_2') & = & dom(M_2) \cup dom(M_2') \end{array}$$

We also have

$dom(M_1) \subseteq dom(M_2)$ and $dom(M_1') \subseteq dom(M_2')$. Hence,
$dom(M_1 \dagger M_1') \subseteq dom(M_2 \dagger M_2')$

if $a \in dom(M_1) \setminus dom(M_1')$, we have $a \in dom(M_2)$ and since $dom(M_1) \cap dom(M_2') = \emptyset$, we also have $a \notin dom(M_2')$. Hence, we get

$$\begin{array}{rcl} (M_1 \dagger M_1')(a) & = & M_1(a) \text{ and} \\ (M_2 \dagger M_2')(a) & = & M_2(a) \end{array}$$

Otherwise, $a \in dom(M_1')$ and we have

$$\begin{array}{rcl}(M_1 \dagger M_1')(a) & = & M_1'(a) \text{ and} \\ (M_2 \dagger M_2')(a) & = & M_2'(a)\end{array}$$

Moreover, we have

$\forall a \in dom(M_1) \setminus dom(M_1').\ (M_1(a), M_2(a)) \in Q$ and
$\forall a \in dom(M_1').\ (M_1'(a), M_2'(a)) \in Q$

Hence, we deduce that

$\forall a \in dom(M_1 \dagger M_1').\ ((M_1 \dagger M_1')(a), (M_2 \dagger M_2')(a)) \in Q$

Let $Q' = \{(M_1(a), M_2(a)) \mid a \in dom(M_1) \wedge M_1 \rhd M_2\}$.

Since $M_1 \rhd M_2$, we deduce easily that $Q' \subseteq \mathcal{F}(Q') \subseteq \mathcal{F}(Q)$.

Let $Q'' = \{(M_1'(a), M_2'(a)) \mid a \in dom(M_1') \wedge M_1' \rhd M_2'\}$.

Since $M_1' \rhd M_2'$, we deduce easily that $Q'' \subseteq \mathcal{F}(Q'') \subseteq \mathcal{F}(Q)$.

This proves that $(M_1 \dagger M_1', M_2 \dagger M_2') \in \overline{\mathcal{F}}$ and consequently $M_1 \dagger M_1' \rhd M_2 \dagger M_2'$. ∎

Lemma 7.8.2 *Let M_1, $M_2 \in \mathbb{M}$, $\sigma : \mathcal{P}_f(V) \times V \to \mathcal{P}_f(V) \times V$ be a substitution.*

We claim that

$$M_1 \rhd M_2 \quad \Rightarrow \quad M_1\sigma \rhd M_2\sigma$$

Proof Let us prove that $(M_1\sigma, M_2\sigma) \in \overline{\mathcal{F}}$.

Let $Q \in \mathcal{P}(\mathcal{U})$ such that

$$\begin{array}{rcl}Q & = & \{(M_1\sigma, M_2\sigma) \mid M_1 \rhd M_2\} \cup \\ & & \{(M_1(a), M_2(a)) \mid a \in dom(M_1) \wedge M_1 \rhd M_2\}\end{array}$$

Let $a \in dom(M_1)$. We have $a \in dom(M_2)$ since $dom(M_1) \subseteq dom(M_2)$.

We have

$$M_1\sigma \quad = \quad [\sigma(a) \mapsto M_1(a) \mid a \in dom(M_1)]$$

Let $b \in dom(M_1\sigma)$. It is clear that $\exists a \in dom(M_1)$ such that $b = \sigma(a)$.

Since $a \in dom(M_1)$ and $dom(M_1) \subseteq dom(M_2)$, we have $\sigma(a) \in dom(M_2\sigma)$.

Hence, $b \in dom(M_2\sigma)$, which means that $dom(M_1\sigma) \subseteq dom(M_2\sigma)$.

It is also clear that $(M_1(a), M_2(a)) \in Q$ and this means that

$\forall b \in dom(M_1\sigma).\ ((M_1\sigma)(b), (M_2\sigma)(b)) \in Q$

This proves that $(M_1\sigma, M_2\sigma) \in \mathcal{F}(Q)$.

Let $Q' = \{(M_1(a), M_2(a)) \mid a \in dom(M_1) \wedge M_1 \rhd M_2\}$.

Since $\forall a \in dom(M_1).\ M_1(a) \rhd M_2(a)$, we have $Q' \subseteq \mathcal{F}(Q') \subseteq \mathcal{F}(Q)$.

Hence, $\forall a \in dom(M_1).\ (M_1(a), M_2(a)) \in \mathcal{F}(Q)$.

Consequently, we have $(M_1\sigma, M_2\sigma) \in \overline{\mathcal{F}}$. ∎

7.8.1 Monotonicity of the Non-Deterministic Choice

Let I be an indexed set and $\theta : \zeta \rightarrow \mathbb{D}$ an environment.

We suppose that $\forall i \in I.\ [\![P_i]\!](\theta) \sqsubseteq [\![P_i']\!](\theta)$.

This means that $\forall i \in I.\ [\![P_i']\!](\theta) \subseteq [\![P_i]\!](\theta)$ and hence we have

$$\bigcup_{i\in I}[\![P_i']\!](\theta) \quad \subseteq \quad \bigcup_{i\in I}[\![P_i]\!](\theta)$$

Finally, we conclude that

$$[\![\oplus_{i\in I} P_i]\!](\theta) \quad \sqsubseteq \quad [\![\oplus_{i\in I} P_i']\!](\theta)$$

7.8.2 Monotonicity of the Strict Sequential Composition

We have to prove that $p\ ;\ q \sqsubseteq p'\ ;\ q'$. This is equivalent to proving that

$$\left\{\begin{array}{l} M_{p;q} \rhd M_{p';q'} \\ \lambda_p \cup \lambda_q \subseteq \lambda_{p'} \cup \lambda_{q'} \\ R_{p;q} \supseteq R_{p';q'} \cup res(r_{p;q}^{-1} r_{p';q'}) \end{array}\right.$$

Case $R_p = \emptyset$

Since $R_p = \emptyset$, we get $p' = p$ using equation 7.2. This means that $M_p = M_{p'}$.

We have to prove that $M_{p;q} \rhd M_{p;q'}$, i.e. $S(M_p, M_q) \rhd S(M_p, M_{q'})$.

The proof uses the co-induction principle that we defined before.

First, if $M_p = [\ ]$, then it is clear that

$$S(M_p, M_q) \quad = \quad M_q \rhd S(M_p, M_{q'}) = M_{q'}$$

Otherwise, let $Q \quad = \quad \{(S(M_1, M_2), S(M_1, M_2')) \mid M_2 \rhd M_2'\} \cup$

$$\{(M_2\sigma_{M_1}, M_2'\sigma_{M_1}) \mid M_2 \rhd M_2'\}$$

Let $(S(M_1, M_2), S(M_1, M_2')) \in Q$. We have to prove that $Q \subseteq \mathcal{F}(Q)$. First, we have to prove that

$dom(S(M_1, M_2)) \subseteq dom(S(M_1, M_2'))$

Let $a \in dom(S(M_1, M_2))$. It is clear from the definition of the function S that $a \in dom(S(M_1, M_2'))$.

Moreover, we have

$$S(M_1, M_2)(a) = \begin{cases} S(M_1(a), M_2), & \text{if } M_1(a) \neq [\,]; \\ M_2\sigma_{M_1}, & \text{Otherwise.} \end{cases}$$

$$S(M_1, M_2')(a) = \begin{cases} S(M_1(a), M_2'), & \text{if } M_1(a) \neq [\,]; \\ M_2'\sigma_{M_1}, & \text{Otherwise.} \end{cases}$$

if $M_1(a) \neq [\,]$, then we have

$(S(M_1(a), M_2), S(M_1(a), M_2')) \in Q$ since $M_1(a) \rhd M_1(a)$

Otherwise, it is clear from the definition of Q that $(M_2\sigma_{M_1}, M_2'\sigma_{M_1}) \in Q$.

This proves that $(S(M_1, M_2), S(M_1, M_2')) \in \mathcal{F}(Q)$.

On the other hand, let $Q' = \{(M_2\sigma_{M_1}, M_2'\sigma_{M_1}) \mid M_2 \rhd M_2'\}$. By lemma 7.8.2, we deduce that $Q' \subseteq \mathcal{F}(Q') \subseteq \mathcal{F}(Q)$ (Since $\mathcal{F}$ is monotone). This proves that $Q \subseteq \mathcal{F}(Q)$.

Since $M_q \rhd M_{q'}$, we have particularly $(S(M_p, M_q), S(M_p, M_{q'})) \in Q$.

Consequently, $(S(M_p, M_q), S(M_p, M_{q'})) \in \overline{F}$ and we conclude that $S(M_p, M_q) \rhd S(M_p, M_{q'})$.

Now, let us prove that the resource constraint is satisfied. First, in this particular case we have

$R_{p;q} = R_q$, $R_{p';q'} = R_{q'}$ and

$R_{p';q'} \cup res(r_{p;q}^{-1} r_{p';q'}) = R_{q'} \cup res(r_{p;q}^{-1} r_{p';q'})$

Since $p = p'$ and using equation 7.1, we get

$$\begin{aligned} res(r_{p;q}^{-1} r_{p';q'}) &= \bigcup_{e \in \xi((r_{p;q})^{-1} r_{p;q'})} res(((\lambda_p \cup \lambda_{q'}) \setminus (\lambda_p \cup \lambda_q))(e)) \\ &= \bigcup_{e \in \xi(r_{p;q'}) \setminus \xi(r_{p;q})} res(((\lambda_p \cup \lambda_{q'}) \setminus (\lambda_p \cup \lambda_q))(e)) \end{aligned}$$

Moreover, we have

$$\begin{aligned} \xi(r_{p;q}) &= dom(\lambda_p \cup \lambda_q) \\ &= dom(\lambda_p) \cup dom(\lambda_q) \\ &= \xi(r_p) \cup \xi(r_q) \end{aligned}$$

$$\begin{aligned}\xi(r_{p;q'}) &= dom(\lambda_p \cup \lambda_{q'}) \\ &= dom(\lambda_p) \cup dom(\lambda_{q'}) \\ &= \xi(r_p) \cup \xi(r_{q'})\end{aligned}$$

Hence, we get

$$\xi(r_{p;q'}) \setminus \xi(r_{p;q}) \quad = \quad (\xi(r_p) \cup \xi(r_{q'})) \setminus (\xi(r_p) \cup \xi(r_q))$$

Here, we recall some laws about set theory: let A, B and C sets, we have

$$\begin{aligned}A \setminus (B \cup C) &= (A \setminus B) \cap (A \setminus C) \\ (A \cup B) \setminus C &= (A \setminus C) \cup (B \setminus C)\end{aligned}$$

Knowing that $\xi(r_{q'}) \cap \xi(r_p) = \emptyset$, $\xi(r_q) \cap \xi(r_p) = \emptyset$, $\xi(r_q) \subseteq \xi(r_{q'})$ and by applying set theory laws, we get

$$\begin{aligned}(\xi(r_p) \cup \xi(r_{q'})) \setminus (\xi(r_p) \cup \xi(r_q)) &= ((\xi(r_p) \cup \xi(r_{q'})) \setminus \xi(r_p)) \cap \\ &\quad ((\xi(r_p) \cup \xi(r_{q'})) \setminus \xi(r_q)) \\ &= ((\xi(r_p) \setminus \xi(r_p)) \cup (\xi(r_{q'}) \setminus \xi(r_p))) \cap \\ &\quad ((\xi(r_p) \setminus \xi(r_q)) \cup (\xi(r_{q'}) \setminus \xi(r_q))) \\ &= \xi(r_{q'}) \cap (\xi(r_p) \cup (\xi(r_{q'}) \setminus \xi(r_q))) \\ &= (\xi(r_{q'}) \cap (\xi(r_{q'}) \setminus \xi(r_q))) \cup \\ &\quad (\xi(r_{q'}) \cap \xi(r_p)) \\ &= \xi(r_{q'}) \cap (\xi(r_{q'}) \setminus \xi(r_q)) \\ &= \xi(r_{q'}) \setminus \xi(r_q)\end{aligned}$$

By the same strategy, we get

$$(\lambda_p \cup \lambda_{q'}) \setminus (\lambda_p \cup \lambda_q) \quad = \quad \lambda_{q'} \setminus \lambda_q$$

Since $res(r_q^{-1} r_{q'}) = \bigcup_{e \in \xi(r_{q'}) \setminus \xi(r_q)} res((\lambda_{q'} \setminus \lambda_q)(e))$, we deduce that

$$res(r_{p;q}^{-1} r_{p;q'}) \quad = \quad res(r_q^{-1} r_{q'})$$

In addition, we have $R_{q'} \cup res(r_{p;q}^{-1} r_{p';q'}) = R_{q'} \cup res(r_q^{-1} r_{q'}) \subseteq R_q$ since $q \sqsubseteq q'$.

Hence, we conclude that $R_{p';q'} \cup res(r_{p;q}^{-1} r_{p';q'}) \subseteq R_{p;q}$.

Case $R_p \neq \emptyset$

We have two subcases:

Subcase $R_{p'} \neq \emptyset$

For this subcase, $r_{p;q} = r_p$, $r_{p';q'} = r_{p'}$ and the first two constraints about the dependence maps and labelling functions are verified since $r_p \preceq r_{p'}$.

For the resource constraint, we have

$R_{p;q} = R_p \cup res(q)$ and

$R_{p';q'} = R_{p'} \cup res(q')$

We also have

$R_{p';q'} \cup res(r_{p';q'}^{-1} r_{p;q}) = R_{p'} \cup res(q') \cup res(r_{p'}^{-1} r_p)$

Besides, since $r_p \preceq r_{p'}$, we have

$R_{p'} \cup res(r_{p'}^{-1} r_p) \subseteq R_p$

Since $q \sqsubseteq q'$, we get $res(q) \supseteq res(q')$ by equation 7.1 and hence we get

$R_{p';q'} \cup res(r_{p';q'}^{-1} r_{p;q}) \subseteq R_p \cup res(q)$

This concludes the proof since $R_p \cup res(q) = R_{p;q}$.

Subcase $R_{p'} = \emptyset$

For this subcase, we already have $M_{p;q} = M_p$. To prove that $M_{p;q} \rhd M_{p';q'}$, it is sufficient to prove that $M_{p'} \rhd M_{p';q'}$. The proof uses co-induction and is similar to the one elaborated for the case $R_p = \emptyset$. In fact, we have just to prove that $S(M_{p'}, [\,]) \rhd S(M_{p'}, M_{q'})$.

Proving that $\lambda_p \cup \lambda_q \subseteq \lambda_{p'} \cup \lambda_{q'}$ is straightforward since $r_p \preceq r_{p'}$ and $r_q \preceq r_{q'}$.

For the resource constraint, we have

$$
\begin{aligned}
R_{p;q} &= R_p \cup res(q) \\
R_{p';q'} &= R_{q'} \\
r_{p;q} &= r_p \text{ since } R_p \neq \emptyset
\end{aligned}
$$

Using the definition of the function res and set theory laws, we can prove that

$$res(r_p^{-1} r_{p';q'}) = res(r_p^{-1} r_{p'}) \cup res(r_{q'})$$

Moreover, we have

$R_p \supseteq res(r_p^{-1} r_{p'})$ and $res(q) \supseteq res(q') = R_{q'} \cup res(r_{q'})$

Hence, we get $R_p \cup res(q) \supseteq R_{q'} \cup res(r_p^{-1} r_{p'}) \cup res(r_{q'})$. Finally, we conclude that

$$R_{p;q} \quad \supseteq \quad R_{p';q'} \cup res(r_{p;q}^{-1} r_{p';q'})$$

7.8.3 Monotonicity of Hiding

We have to prove that $p \backslash R \sqsubseteq p' \backslash R$. This is equivalent to proving that

$$\begin{cases} M_{p\backslash R} \rhd M_{p'\backslash R} \\ \lambda_{p\backslash R} \subseteq \lambda_{p'\backslash R} \\ R_{p\backslash R} \supseteq R_{p'\backslash R} \cup res(r_{p\backslash R}^{-1} r_{p'\backslash R}) \end{cases}$$

Proving that $M_{p\backslash R} \rhd M_{p'\backslash R}$ is equivalent to proving that $H_R(M_p) \rhd H_R(M_{p'})$.

Let $Q \quad = \quad \{(H_R(M_1), H_R(M_1')) \mid M_1 \rhd M_1'\}$.

We have to prove that $Q \subseteq \mathcal{F}(Q)$.

Let $(H_R(M_1), H_R(M_1')) \in Q$. First, let us prove that

$dom(H_R(M_1)) \subseteq dom(H_R(M_1'))$

Let $a \in dom(H_R(M_1))$, then $\exists b \in dom(M_1)$ such that $a = (\sigma^+(\pi_1(b)), \pi_2(b))$. Since $M_1 \rhd M_1'$, $dom(M_1) \subseteq dom(M_1')$. This means that $b \in dom(M_1')$ and hence $(\sigma^+(\pi_1(b)), \pi_2(b)) \in dom(H_R(M_1'))$. Therefore, we deduce that $a \in dom(H_R(M_1'))$. This proves that $dom(H_R(M_1)) \subseteq dom(H_R(M_1'))$.

We claim that $\forall a \in dom(H_R(M_1))$. $(H_R(M_1)(a), H_R(M_1')(a)) \in \mathcal{F}(Q)$. In fact, let $a \in dom(H_R(M_1))$, then $\exists b \in dom(M_1)$ such that $a = (\sigma^+(\pi_1(b)), \pi_2(b))$. Moreover, we have

$$\begin{array}{lcl} H_R(M_1)(a) & = & H_R(M_1(b)) \text{ and} \\ H_R(M_1')(a) & = & H_R(M_1'(b)) \end{array}$$

Since $M_1 \rhd M_1'$, we have $M_1(b) \rhd M_1'(b)$ and $(H_R(M_1(b)), H_R(M_1'(b))) \in Q$. Therefore, we deduce that $(H_R(M_1)(a), H_R(M_1)(a)) \in \mathcal{F}(Q)$. This proves that $Q \subseteq \mathcal{F}(Q)$. Thus, we conclude that $H_R(M_1) \rhd H_R(M_1')$.

Now, since $M_p \rhd M_{p'}$, we have particularly

$(H_R(M_p), H_R(M_{p'})) \in Q \subseteq \mathcal{F}(Q)$

Consequently, $H_R(M_p) \rhd H_R(M_{p'})$.

For the labelling constraint, we have

$$\lambda_{p\backslash R} \quad = \quad (\lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}) \cup \{(\sigma(e), \eta(\lambda_p(e))) \mid e \in V_R^p\} \text{ and}$$

$$\lambda_{p'\backslash R} \quad = \quad (\lambda_{p'} \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\}) \cup \{(\sigma(e), \eta(\lambda_{p'}(e))) \mid e \in V_R^{p'}\}$$

We also have

$$\begin{aligned} V_R^p &= \{e \in \xi(r_p) \mid res(\lambda_p(e)) \subseteq R\} \text{ and} \\ V_R^{p'} &= \{e \in \xi(r_{p'}) \mid res(\lambda_{p'}(e)) \subseteq R\} \end{aligned}$$

Since $\xi(r_p) \subseteq \xi(r_{p'})$, it is clear that $V_R^p \subseteq V_R^{p'}$. Hence, it is straightforward to prove that

$$\{(\sigma(e, \eta(\lambda_p(e))) \mid e \in V_R^p\} \subseteq \{(\sigma(e), \eta(\lambda_{p'}(e))) \mid e \in V_R^{p'}\} \tag{7.10}$$

Moreover, since $\lambda_p \subseteq \lambda_{p'}$, we have

$$\begin{aligned} \lambda_{p'} \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} &= (\lambda_p \cup (\lambda_{p'} \setminus \lambda_p)) \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \\ &= (\lambda_p \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\}) \cup \\ &\quad ((\lambda_{p'} \setminus \lambda_p) \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\}) \end{aligned} \tag{7.11}$$

Using set theory laws, we also have

$$\begin{aligned} &\lambda_p \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} = \\ &\lambda_p \setminus (\{(e, \lambda_p(e)) \mid e \in V_R^p\} \cup \\ &\qquad (\{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\})) = \\ &(\lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}) \cap \\ &(\lambda_p \setminus (\{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\})) \end{aligned}$$

Since it is clear that

$$\lambda_p \cap (\{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}) = \emptyset$$

We get

$$\lambda_p \setminus (\{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}) = \lambda_p$$

Moreover, since $\lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\} \subseteq \lambda_p$, we deduce that

$$\lambda_p \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} = \lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\}$$

Using 7.11, we deduce that

$$\lambda_{p'} \setminus \{(e, \lambda_{p'}(e)) \mid e \in V_R^{p'}\} \supseteq \lambda_p \setminus \{(e, \lambda_p(e)) \mid e \in V_R^p\} \tag{7.12}$$

Using 7.10 and 7.12 we conclude that $\lambda_{p \setminus R} \subseteq \lambda_{p' \setminus R}$.

For the resource constraint, we have to prove that

$R_{p\backslash R} \supseteq R_{p'\backslash R} \cup res(r^{-1}_{p\backslash R} r_{p'\backslash R})$

By the definition of the operator $\backslash$, we have $R_{p\backslash R} = R_p$ and $R_{p'\backslash R} = R_{p'}$.

We have

$$res(r^{-1}_{p\backslash R} r_{p'\backslash R}) \quad = \quad \bigcup_{e \in \xi(r_{p'\backslash R}) \backslash \xi(r_{p\backslash R})} res((\lambda_{p'\backslash R} \setminus \lambda_{p\backslash R})(e))$$

We also have

$$\begin{aligned}
\xi(r_{p'\backslash R}) &= dom((\lambda_{p'} \setminus \{(e, \lambda_{p'}(e)) \mid e \in V^{p'}_R\}) \cup \{(\sigma(e), \eta(\lambda_{p'}(e))) \mid e \in V^{p'}_R\}) \\
&= (dom(\lambda_{p'}) \setminus V^{p'}_R) \cup \{\sigma(e) \mid e \in V^{p'}_R\} \\
&= (\xi(r_{p'}) \setminus V^{p'}_R) \cup \{\sigma(e) \mid e \in V^{p'}_R\} \\
&= \sigma^+(\xi(r_{p'}))
\end{aligned}$$

Moreover, we have

$$\begin{aligned}
\xi(r_{p\backslash R}) &= (\xi(r_p) \setminus V^p_R) \cup \{\sigma(e) \mid e \in V^p_R\} \\
&= \sigma^+(\xi(r_p))
\end{aligned}$$

Now, let $e \in \xi(r_{p'}) \setminus \xi(r_p)$. It is easy to prove that $\sigma(e) \in \xi(r_{p'\backslash R}) \setminus \xi(r_{p\backslash R})$ using the above equations.

This means that $e \in dom(\lambda_{p'}) \setminus dom(\lambda_p)$ and $\sigma(e) \in dom(\lambda_{p'\backslash R}) \setminus dom(\lambda_{p\backslash R})$.

If $res((\lambda_{p'} \setminus \lambda_p)(e)) \subseteq R$, then knowing that $\lambda_{p\backslash R} \subseteq \lambda_{p'\backslash R}$, we get

$$\begin{aligned}
res((\lambda_{p'\backslash R} \setminus \lambda_{p\backslash R})(\sigma(e))) &= res(\lambda_{p'\backslash R}(\sigma(e))) \\
&= res(\eta(\lambda_{p'}(e))) \\
&= res(\lambda_{p'}(e)) \\
&= res((\lambda_{p'} \setminus \lambda_p)(e))
\end{aligned}$$

Otherwise, if $res((\lambda_{p'} \setminus \lambda_p)(e)) \nsubseteq R$, then $e \in \xi(r_{p'\backslash R}) \setminus \xi(r_{p\backslash R})$ and we have

$$\begin{aligned}
res((\lambda_{p'\backslash R} \setminus \lambda_{p\backslash R})(\sigma(e))) &= res((\lambda_{p'\backslash R} \setminus \lambda_{p\backslash R})(e)) \\
&= res(\lambda_{p'\backslash R}(e)) \\
&= res(\lambda_{p'}(e)) \\
&= res((\lambda_{p'} \setminus \lambda_p)(e))
\end{aligned}$$

Hence, we have

$\bigcup_{e\in\xi(r_{p'})\setminus\xi(r_p)} res((\lambda_{p'} \setminus \lambda_p)(e)) \subseteq \bigcup_{e'\in\xi(r_{p'\setminus R})\setminus\xi(r_{p\setminus R})} res((\lambda_{p'\setminus R} \setminus \lambda_{p\setminus R})(e'))$

Since it is clear that σ is surjective, we can use the same strategy to prove that

$\bigcup_{e'\in\xi(r_{p'\setminus R})\setminus\xi(r_{p\setminus R})} res((\lambda_{p'\setminus R} \setminus \lambda_{p\setminus R})(e')) \subseteq \bigcup_{e\in\xi(r_{p'})\setminus\xi(r_p)} res((\lambda_{p'} \setminus \lambda_p)(e))$

Hence, it is clear that $res(r^{-1}_{p\setminus R} r_{p'\setminus R}) = res(r_p^{-1} r_{p'})$.

Since $p \sqsubseteq p'$, we have $R_p \supseteq R_{p'} \cup res(r_p^{-1} r_{p'})$. Thus, the resource constraint is satisfied.

7.8.4 Monotonicity of the Weak Sequential Composition

Hereafter, we prove that $\circ$ is monotone. This means that we have to prove that

$$\begin{cases} r_p \bullet r_{\mu^q_{R_p}} \preceq r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}} \\ \\ R_p \cup R_q \cup res(r^{-1}_{\mu^q_{R_p}} r_q) \supseteq \\ R_{p'} \cup R_{q'} \cup res(r^{-1}_{\mu^{q'}_{R_{p'}}} r_{q'}) \cup res((r_{p\circ q})^{-1}(r_{p'\circ q'})) \end{cases}$$

First, we prove that $r_p \bullet r_{\mu^q_{R_p}} \preceq r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}$.

This is equivalent to proving that

$$\begin{cases} \Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \ddagger \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) \rhd \\ \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \ddagger \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}) \\ \\ \lambda_p \cup \lambda_{\mu^q_{R_p}} \subseteq \lambda_{p'} \cup \lambda_{\mu^{q'}_{R_{p'}}} \end{cases}$$

Note that to lighten the notation, $\Phi_{p,\mu^q_{R_p}}$ is denoted by Φ and $\Phi_{p',\mu^{q'}_{R_{p'}}}$ is denoted by Φ' in the above constraints.

We have

$$\Phi = \{(e,e') \in \xi(r_p) \times \xi(r_{\mu^q_{R_p}}) \mid res(\lambda_p(e)) \cap res(\lambda_{\mu^q_{R_p}}(e')) \neq \emptyset\}$$

$$\Phi' = \{(e,e') \in \xi(r_{p'}) \times \xi(r_{\mu^{q'}_{R_{p'}}}) \mid res(\lambda_{p'}(e)) \cap res(\lambda_{\mu^{q'}_{R_{p'}}}(e')) \neq \emptyset\}$$

We claim that

$\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$ and

$\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \rhd \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))$

We recall that

$$\begin{aligned}\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) &= [(\pi_1(b) \cup \{a \in \xi(r_p) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mapsto \\ &\quad \Lsh_{\Phi} (M_{\mu^q_{R_p}}(b), r_p) \mid b \in dom(M_{\mu^q_{R_p}})]\end{aligned}$$

$$\begin{aligned}\Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}) &= [(\pi_1(b) \cup \{a \in \xi(r_{p'}) \mid (a, \pi_2(b)) \in \Phi'\}, \pi_2(b)) \mapsto \\ &\quad \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}(b), r_{p'}) \mid b \in dom(M_{\mu^{q'}_{R_{p'}}})]\end{aligned}$$

The proof steps are the following:

1. We have to prove that $\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$.

   Let

   $$\begin{aligned}Q \;=\; &\{(\Lsh_{\Phi} (M_2, r_1), \Lsh_{\Phi'} (M'_2, r'_1)) \mid \pi_1(r_1) \rhd \pi_1(r'_1) \wedge M_2 \rhd M'_2 \wedge \Phi \subseteq \Phi' \wedge \\ &\forall b \in \xi(r_2).\ \forall e \in \xi(r'_1) \setminus \xi(r_1).\ (e, b) \notin \Phi'\}\end{aligned}$$

   We claim that $Q \subseteq \mathcal{F}(Q)$.

   Let $(\Lsh_{\Phi} (M_2, r_1), \Lsh_{\Phi'} (M'_2, r'_1)) \in Q$.

   First, we have to prove that $dom(\Lsh_{\Phi} (M_2, r_1)) \subseteq dom(\Lsh_{\Phi'} (M'_2, r'_1))$.

   Let $b \in dom(M_2)$. Since $\Phi \subseteq \Phi'$, we have

   $\forall e \in \xi(r'_1) \setminus \xi(r_1).\ (e, \pi_2(b)) \notin \Phi$

   Hence, $\forall b \in dom(M_2)$, we have

   $\{a \in \xi(r'_1) \mid (a, \pi_2(b)) \in \Phi'\} = \{a \in \xi(r_1) \mid (a, \pi_2(b)) \in \Phi\}$

   This means that $\forall b \in dom(M_2)$, we have

   $(\pi_1(b) \cup \{a \in \xi(r'_1) \mid (a, \pi_2(b)) \in \Phi'\}, \pi_2(b)) =$
   $(\pi_1(b) \cup \{a \in \xi(r_1) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b))$

   This proves that $dom(\Lsh_{\Phi} (M_2, r_1)) \subseteq dom(\Lsh_{\Phi'} (M'_2, r'_1))$.

   From the definition of the operator $\Lsh$, it follows that

   $\forall a \in dom(\Lsh_{\Phi} (M_2, r_1)).\ \exists b \in dom(M_2).\ \Lsh_{\Phi} (M_2, r_1)(a) = \Lsh_{\Phi} (M_2(b), r_1)$

   Moreover, we have

   $\Lsh_{\Phi'} (M'_2, r'_1)(a) = \Lsh_{\Phi'} (M'_2(b), r'_1)$

   Since $M_2 \rhd M'_2$, we also have $M_2(b) \rhd M'_2(b)$.

   This means that $(\Lsh_{\Phi} (M_2(b), r_1), \Lsh_{\Phi'} (M'_2(b), r'_1)) \in Q$.

   Consequently, $(\Lsh_{\Phi} (M_2, r_1)(a), \Lsh_{\Phi'} (M'_2, r'_1)(a)) \in Q$, which proves that $Q \subseteq \mathcal{F}(Q)$.

   Now, we claim that $r_{\mu^q_{R_p}} \preceq r_{\mu^{q'}_{R_{p'}}}$. We have

$$\begin{array}{lcl} \mu^q_{R_p} & = & \sqcup\{t \sqsubseteq q \mid res(\pi_1(t)) \cap R_p = \emptyset\} \\ \mu^{q'}_{R_{p'}} & = & \sqcup\{t \sqsubseteq q' \mid res(\pi_1(t)) \cap R_{p'} = \emptyset\} \end{array}$$

It is easy to show that

$\mu^q_{R_p} \in \{t \sqsubseteq q \mid res(\pi_1(t)) \ \cap \ R_p = \emptyset\}$ and

$\mu^{q'}_{R_{p'}} \in \{t \sqsubseteq q' \mid res(\pi_1(t)) \ \cap \ R_{p'} = \emptyset\}$

In fact, it is easy to prove that $res(\mu^q_{R_p}) \cap R_p = \emptyset$ using the fact that $res(u) \cap R_p = \emptyset$ for $u \in \{t \sqsubseteq q \mid res(\pi_1(t)) \cap R_p = \emptyset\}$.

Since $R_p \supseteq R_{p'}$ and $q \sqsubseteq q'$, we have

$\forall u \in \{t \sqsubseteq q \mid res(\pi_1(t)) \cap R_p = \emptyset\}.\ \exists u' \in \{t \sqsubseteq q' \mid res(\pi_1(t)) \cap R_{p'} = \emptyset\}$ such that $u \sqsubseteq u'$.

Hence, $\mu^q_{R_p} \sqsubseteq \mu^{q'}_{R_{p'}}$, which means that $r_{\mu^q_{R_p}} \preceq r_{\mu^{q'}_{R_{p'}}}$.

Consequently, we have $M_{\mu^q_{R_p}} \rhd M_{\mu^{q'}_{R_{p'}}}$.

We also have

$\forall b \in \xi(r_{\mu^q_{R_p}}).\ \forall e \in \xi(r_{p'}) \setminus \xi(r_p).\ (e, b) \notin \Phi'$

In fact, we have

$\forall b \in \xi(r_{\mu^q_{R_p}}).\ res(\lambda_{\mu^q_{R_p}}(b)) \cap R_p = \emptyset$ and

$R_p \supseteq res(r_p^{-1} r_{p'}) = \bigcup_{e \in \xi(r_{p'}) \setminus \xi(r_p)} res((\lambda_{p'} \setminus \lambda_p)(e))$

This means that

$\forall b \in \xi(r_{\mu^q_{R_p}}).\ \forall e \in \xi(r_{p'}) \setminus \xi(r_p).\ (e, b) \notin \Phi'$

Moreover, since $\xi(r_p) \subseteq \xi(r_{p'})$ and $\xi(r_{\mu^q_{R_p}}) \subseteq \xi(r_{\mu^{q'}_{R_{p'}}})$, we have $\Phi \subseteq \Phi'$.

Hence, it is clear that we have particularly

$(\Lsh_\Phi (M_{\mu^q_{R_p}}, r_p), \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \in Q \subseteq \mathcal{F}(Q)$

Finally, we deduce that $\Lsh_\Phi (M_{\mu^q_{R_p}}, r_p) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$.

2. We have to prove that

$\forall a \in dom(M_p).\ Succ(a, \Lsh_\Phi (M_{\mu^q_{R_p}}, r_p), \Phi) \ \rhd Succ(a, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}), \Phi')$

We recall that

$$\begin{array}{lcl} Succ(a, M, \Phi) & = & [b \mapsto M(b) \mid b \ \in \ dom(M) \wedge (\pi_2(a), \pi_2(b)) \in \Phi] \dagger \\ & & \dagger_{b \in dom(M)} Succ(a, M(b), \Phi) \end{array}$$

Let $\Psi(a, M, \Phi) = [b \mapsto M(b) \mid b \ \in \ dom(M) \wedge (\pi_2(a), \pi_2(b)) \ \in \ \Phi]$.

Since $\Lsh_\Phi (M_{\mu^q_{R_p}}, r_p) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$ and $\Phi \subseteq \Phi'$, it is easy to prove that

$\Psi(a, \Lsh_\Phi (M_{\mu^q_{R_p}}, r_p), \Phi) \rhd \Psi(a, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}), \Phi')$

Let

$$Q = \{(Succ(a, \Lsh_\Phi (M_2, r_1)(b), \Phi), Succ(a, \Lsh_{\Phi'} (M_2', r_1')(b), \Phi')) \mid a \in dom(\pi_1(r_1)) \wedge b \in dom(\Lsh_\Phi (M_2, r_1)) \wedge \pi_1(r_1) \rhd \pi_1(r_1') \wedge M_2 \rhd M_2' \wedge \Phi \subseteq \Phi'\} \cup$$

$$\{(\Lsh_\Phi (M_2, r_1), \Lsh_{\Phi'} (M_2', r_1')) \mid \pi_1(r_1) \rhd \pi_1(r_1') \wedge M_2 \rhd M_2' \wedge \Phi \subseteq \Phi' \wedge \forall b \in \xi(r_2).\ \forall e \in \xi(r_1') \setminus \xi(r_1).\ (e, b) \notin \Phi'\}$$

We have to prove that $Q \subseteq \mathcal{F}(Q)$.

Let

$$Q' = \{(\Lsh_\Phi (M_2, r_1), \Lsh_{\Phi'} (M_2', r_1')) \mid \pi_1(r_1) \rhd \pi_1(r_1') \wedge M_2 \rhd M_2' \wedge \Phi \subseteq \Phi' \wedge \forall b \in \xi(r_2).\ \forall e \in \xi(r_1') \setminus \xi(r_1).(e, b) \notin \Phi'\}$$

We know that $Q' \subseteq \mathcal{F}(Q')$, hence

$\forall c \in dom(\Lsh_\Phi (M_2, r_1)(b)).\ (\Lsh_\Phi (M_2, r_1)(b)(c), \Lsh_{\Phi'} (M_2', r_1')(b)(c)) \in Q' \subseteq Q$

Particularly, $\forall c \in dom(\Lsh_\Phi (M_2, r_1)(b))$ such that $(\pi_2(a), \pi_2(c)) \in \Phi$, we have

$Succ(a, \Lsh_\Phi (M_2, r_1)(b), \Phi)(c) = \Lsh_\Phi (M_2, r_1)(b)(c)$

Since $dom(\Lsh_\Phi (M_2, r_1)(b)) \subseteq dom(\Lsh_\Phi (M_2', r_1')(b))$, we have

$Succ(a, \Lsh_{\Phi'} (M_2', r_1')(b), \Phi')(c) = \Lsh_{\Phi'} (M_2', r_1')(b)(c)$

Hence, we conclude that

$(Succ(a, \Lsh_\Phi (M_2, r_1)(b), \Phi)(c), Succ(a, \Lsh_{\Phi'} (M_2', r_1')(b), \Phi')(c)) \in Q$

This concludes the proof that $Q \subseteq \mathcal{F}(Q)$.

Now, since it is clear that $\varphi(\Lsh_\Phi (M_2, r_1))$ and $\varphi(\Lsh_{\Phi'} (M_2', r_1'))$ are disjoint, we deduce that we have particularly $dom(\Lsh_\Phi (M_2, r_1))$ and $dom(\Lsh_{\Phi'} (M_2', r_1'))$ are disjoint. Hence, by lemma 7.8.1 (the lemma is also applicable for a countable set of arguments), we deduce that

$\dagger_{b \in dom(\Lsh_\Phi(M_2, r_1))} Succ(a, \Lsh_\Phi (M_2, r_1)(b), \Phi) \rhd$
$\dagger_{b \in dom(\Lsh_{\Phi'}(M_2', r_1'))} Succ(a, \Lsh_{\Phi'} (M_2', r_1')(b), \Phi')$

Knowing that $\Psi(a, \Lsh_\Phi (M_2, r_1), \Phi) \rhd \Psi(a, \Lsh_{\Phi'} (M_2', r_1'), \Phi')$, again by lemma 7.8.1, we deduce that $Succ(a, \Lsh_\Phi (M_2, r_1), \Phi) \rhd Succ(a, \Lsh_{\Phi'} (M_2', r_1'), \Phi')$.

Hence, since $M_{\mu^q_{R_p}} \rhd M_{\mu^{q'}_{R_{p'}}}$, $M_p \rhd M_{p'}$ and

$\forall b \in \xi(r_{\mu^q_{R_p}}).\ \forall e \in \xi(r_{p'}) \setminus \xi(r_p).\ (e, b) \notin \Phi'$, we conclude that we have particularly

$Succ(a, \Lsh_\Phi (M_{\mu^q_{R_p}}, r_p), \Phi) \rhd Succ(a, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}), \Phi')$

3. We have to prove that $\Rsh_\Phi (M_p, \Lsh_\Phi (M_{\mu^q_{R_p}}, r_p)) \rhd \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))$.

Let

$$
\begin{aligned}
Q \quad = \quad & \{(\Rsh_{\Phi}(M_1, M_2), \Rsh_{\Phi'}(M_1', M_2')) \mid M_1 \rhd M_1' \wedge M_2 \rhd M_2'\} \;\cup \\
& \{(\Rsh_{\Phi}(M_1(a), M_2) \;\dagger\; Succ(a, M_2, \Phi), \Rsh_{\Phi'}(M_1'(a), M_2') \;\dagger\; Succ(a, M_2', \Phi')) \mid \\
& \;\; a \in dom(M_1) \wedge M_1 \rhd M_1' \wedge M_2 \rhd M_2'\} \;\cup \\
& \{(Succ(a, M_2, \Phi), Succ(a, M_2', \Phi')) \mid M_2 \rhd M_2' \wedge \Phi \subseteq \Phi'\}
\end{aligned}
$$

We have to prove that $Q \subseteq \mathcal{F}(Q)$.

Let $(\Rsh_{\Phi}(M_1, M_2), \Rsh_{\Phi'}(M_1', M_2')) \in Q$. From the definition of the operator $\Rsh$, it follows that

$dom(\Rsh_{\Phi}(M_1, M_2)) \subseteq dom(\Rsh_{\Phi'}(M_1', M_2'))$ since $dom(M_1) \subseteq dom(M_1')$

Moreover, we have

$\forall a \in dom(\Rsh_{\Phi}(M_1, M_2)).\ \Rsh_{\Phi}(M_1, M_2)(a) \;= \Rsh_{\Phi}(M_1(a), M_2) \;\dagger\; Succ(a, M_2, \Phi)$

Since $a \in dom(\Rsh_{\Phi'}(M_1', M_2'))$, we also have

$\Rsh_{\Phi'}(M_1', M_2')(a) \;= \Rsh_{\Phi'}(M_1'(a), M_2') \;\dagger\; Succ(a, M_2', \Phi')$

This means that $(\Rsh_{\Phi}(M_1, M_2)(a), \Rsh_{\Phi'}(M_1', M_2')(a)) \in Q$.

Consequently, $(\Rsh_{\Phi}(M_1, M_2), \Rsh_{\Phi'}(M_1', M_2')) \in \mathcal{F}(Q)$.

Besides, we have

$$
(\Rsh_{\Phi}(M_1(a), M_2) \;\dagger\; Succ(a, M_2, \Phi))(b) = \\
\begin{cases} \Rsh_{\Phi}(M_1(a), M_2)(b), & \text{if } b \in dom(\Rsh_{\Phi}(M_1(a), M_2)); \\ Succ(a, M_2, \Phi)(b), & \text{if } b \in dom(M_2). \end{cases}
$$

We also have

$$
(\Rsh_{\Phi'}(M_1'(a), M_2') \;\dagger\; Succ(a, M_2', \Phi'))(b) = \\
\begin{cases} \Rsh_{\Phi'}(M_1'(a), M_2')(b)\ , & \text{if } b \in dom(\Rsh_{\Phi'}(M_1'(a), M_2')); \\ Succ(a, M_2', \Phi')(b), & \text{if } b \in dom(M_2'). \end{cases}
$$

Let $Q' = \{(Succ(a, M_2, \Phi), Succ(a, M_2', \Phi')) \mid M_2 \rhd M_2' \wedge \Phi \subseteq \Phi'\}$.

We know that $\forall b \in dom(\Rsh_{\Phi}(M_1(a), M_2)).\ (\Rsh_{\Phi}(M_1(a), M_2)(b), \Rsh_{\Phi'}(M_1'(a), M_2')(b)) \in Q$.

Moreover, since $(Succ(a, M_2, \Phi), Succ(a, M_2', \Phi')) \in \mathcal{F}(Q') \subseteq \mathcal{F}(Q)$, we have

$\forall b \in dom(M_2).\ (Succ(a, M_2, \Phi)(b), Succ(a, M_2', \Phi')(b)) \in Q$

This means that

$(\Rsh_{\Phi}(M_1(a), M_2) \;\dagger\; Succ(a, M_2, \Phi), \Rsh_{\Phi'}(M_1'(a), M_2') \;\dagger\; Succ(a, M_2', \Phi')) \in \mathcal{F}(Q)$

This concludes the proof that $Q \subseteq \mathcal{F}(Q)$.

Now, it is clear that we have particularly

$(\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)), \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))) \in Q$

Hence, we conclude that

$\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \rhd \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))$

4. We have to prove that

$\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \ddagger \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) \rhd$
$\Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \ddagger \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$

We have

$\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \ddagger \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) =$
$\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \dagger$
$[a \mapsto \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)(a) \mid a \in dom(\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \wedge \pi_1(a) = \emptyset]$ and

$\Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \ddagger \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}) =$
$\Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \dagger$
$[a \mapsto \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})(a) \mid a \in dom(\Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \wedge \pi_1(a) = \emptyset]$

Since $\Rsh_{\Phi} (M_p, \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \rhd \Rsh_{\Phi'} (M_{p'}, \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))$, we only have to prove that

$[a \mapsto \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)(a) \mid a \in dom(\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \wedge \pi_1(a) = \emptyset] \rhd$
$[a \mapsto \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})(a) \mid a \in dom(\Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \wedge \pi_1(a) = \emptyset]$

After that, we use lemma 7.8.1 to conclude the proof.

Let $a \in dom(\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p))$ such that $\pi_1(a) = \emptyset$.

Since $\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})$, it is clear that $a \in dom(\Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'}))$.

Since we have $\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)(a) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})(a)$, we conclude easily that

$[a \mapsto \Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)(a) \mid a \in dom(\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_p)) \wedge \pi_1(a) = \emptyset] \rhd$
$[a \mapsto \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})(a) \mid a \in dom(\Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{p'})) \wedge \pi_1(a) = \emptyset]$

This concludes the proof about the prefix constraint.

Proving that $\lambda_p \cup \lambda_{\mu^q_{R_p}} \subseteq \lambda_{p'} \cup \lambda_{\mu^{q'}_{R_{p'}}}$ is straightforward since $r_p \preceq r_{p'}$ and $r_{\mu^q_{R_p}} \preceq r_{\mu^{q'}_{R_{p'}}}$.

For the resource constraint, from the definition of the semantic function $\circ$ we get

$res(r^{-1}_{p \circ q} r_{p' \circ q'}) \quad = \quad res((r_p \bullet r_{\mu^q_{R_p}})^{-1}(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}))$

Now, we have to prove that

$$(7.13)\qquad \begin{aligned} R_p \cup R_q \cup res(r^{-1}_{\mu^q_{R_p}} r_q) \quad \supseteq \quad & R_{p'} \cup R_{q'} \cup res(r^{-1}_{\mu^{q'}_{R_{p'}}} r_{q'}) \cup \\ & res((r_p \bullet r_{\mu^q_{R_p}})^{-1}(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}})) \end{aligned}$$

We have

$$\begin{aligned} \xi(r_p \bullet r_{\mu^q_{R_p}}) &= dom(\lambda_p \cup \lambda_{\mu^q_{R_p}}) \\ &= dom(\lambda_p) \cup dom(\lambda_{\mu^q_{R_p}}) \\ &= \xi(r_p) \cup \xi(r_{\mu^q_{R_p}}) \end{aligned}$$

Besides, we have

$$\xi(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}) = \xi(r_{p'}) \cup \xi(r_{\mu^{q'}_{R_{p'}}})$$

Knowing that $\xi(r_{p'}) \cap \xi(r_{\mu^q_{R_p}}) = \emptyset$ and $\xi(r_p) \cap \xi(r_{\mu^{q'}_{R_{p'}}}) = \emptyset$ and using set theory laws (we omit details), we have

$$\begin{aligned} \xi((r_p \bullet r_{\mu^q_{R_p}})^{-1}(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}})) &= \xi(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_p \bullet r_{\mu^q_{R_p}}) \\ &= \xi(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_p \bullet r_{\mu^q_{R_p}}) \\ &= (\xi(r_{p'}) \cup \xi(r_{\mu^{q'}_{R_{p'}}})) \setminus (\xi(r_p) \cup \xi(r_{\mu^q_{R_p}})) \\ &= (\xi(r_{p'}) \setminus \xi(r_p)) \cup (\xi(r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_{\mu^q_{R_p}})) \end{aligned}$$

Using the same strategy, it is easy to show that

$$(\lambda_{p'} \cup \lambda_{\mu^{q'}_{R_{p'}}}) \setminus (\lambda_p \cup \lambda_{\mu^q_{R_p}}) = (\lambda_{p'} \setminus \lambda_p) \cup (\lambda_{\mu^{q'}_{R_{p'}}} \setminus \lambda_{\mu^q_{R_p}})$$

Hence, we get

$$\begin{aligned} & res((r_p \bullet r_{\mu^q_{R_p}})^{-1}(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}})) && = \\ & \bigcup_{e \in \xi((r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}}) \setminus (r_p \bullet r_{\mu^q_{R_p}}))} res(((\lambda_{p'} \cup \lambda_{\mu^{q'}_{R_{p'}}}) \setminus (\lambda_p \cup \lambda_{\mu^q_{R_p}}))(e)) && = \\ & \bigcup_{e \in (\xi(r_{p'}) \setminus \xi(r_p)) \cup (\xi(r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_{\mu^q_{R_p}}))} res(((\lambda_{p'} \setminus \lambda_p) \cup (\lambda_{\mu^{q'}_{R_{p'}}} \setminus \lambda_{\mu^q_{R_p}}))(e)) \end{aligned}$$

Now, it is easy to prove that $\forall p, q \in \mathbb{C}$ such that $dom(\lambda_p)$ and $dom(\lambda_q)$ are disjoint, we have

$$(7.14)\qquad res(r_p) \cup res(r_q) = \bigcup_{e \in \xi(r_p) \cup \xi(r_q)} res((\lambda_p \cup \lambda_q)(e)) \text{ where}$$

$$(\lambda_p \cup \lambda_q)(e) = \begin{cases} \lambda_p(e), & \text{if } e \in \xi(r_p); \\ \lambda_q(e), & \text{Otherwise.} \end{cases}$$

It is obvious that the two processes $p^{-1}p'$ and $(\mu^q_{R_p})^{-1}\mu^{q'}_{R_{p'}}$ are labelled differently. Hence, using equation 7.14, we get

$$\begin{aligned} res((r_p \bullet r_{\mu^q_{R_p}})^{-1}(r_{p'} \bullet r_{\mu^{q'}_{R_{p'}}})) &= \bigcup_{e\in\xi(r_{p'})\setminus\xi(r_p)} res((\lambda_{p'} \setminus \lambda_p)(e)) \cup \\ &\quad \bigcup_{e\in\xi(r_{\mu^{q'}_{R_{p'}}})\setminus\xi(r_{\mu^q_{R_p}})} res((\lambda_{\mu^{q'}_{R_{p'}}} \setminus \lambda_{\mu^q_{R_p}})(e)) \\ &= res(r_p{}^{-1} r_{p'}) \cup res(r_{\mu^q_{R_p}}{}^{-1} r_{\mu^{q'}_{R_{p'}}}) \end{aligned}$$

We have $res(r_{\mu^q_{R_p}}{}^{-1} r_{\mu^{q'}_{R_{p'}}}) \cup res(r_{\mu^{q'}_{R_{p'}}}{}^{-1} r_{q'}) = res(r_{\mu^q_{R_p}}{}^{-1} r_{q'})$.

We also have $res(r^{-1}_{\mu^q_{R_p}} r_{q'}) = res(r_{\mu^q_{R_p}}{}^{-1} r_q) \cup res(r_q^{-1} r_{q'})$.

Since $R_q \supseteq R_{q'} \cup res(r_q^{-1} r_{q'})$, we deduce that

$$R_q \cup res(r_{\mu^q_{R_p}}{}^{-1} r_q) \supseteq R_{q'} \cup res(r_q^{-1} r_{q'}) \cup res(r_{\mu^q_{R_p}}{}^{-1} r_q) \tag{7.15}$$

We also have

$$R_p \supseteq R_{p'} \cup res(r_p^{-1} r_{p'}) \tag{7.16}$$

Hence, the constraint (7.13) is satisfied by taking the union of the constraints (7.15) and (7.16).

7.8.5 Monotonicity of the Parallel Composition

Hereafter, we prove the monotonicity of the parallel composition operator $\|$.

To lighten the notation, $\Psi_{\mu^p_{R_q}, \mu^q_{R_p}}$ is denoted by Ψ and $\Psi_{\mu^{p'}_{R_{q'}}, \mu^{q'}_{R_{p'}}}$ is denoted by Ψ'. Moreover, we remove the index from $\Phi_i \in \Psi$.

Let $p \sqsubseteq p'$ and $q \sqsubseteq q'$. We are dealing with the case where

$R_{\mu^p_{R_q}} \cap res(r_{\mu^q_{R_p}}) = \emptyset$ and $R_{\mu^q_{R_p}} \cap res(r_{\mu^p_{R_q}}) = \emptyset$

and also

$R_{\mu^{p'}_{R_{q'}}} \cap res(r_{\mu^{q'}_{R_{p'}}}) = \emptyset$ and $R_{\mu^{q'}_{R_{p'}}} \cap res(r_{\mu^{p'}_{R_{q'}}}) = \emptyset$

The monotonicity in the other case is obvious since the semantic function is constant.

We recall that

$$
\begin{array}{lcl}
p \parallel q & = & \bigcup_{\Phi\in\Psi} \uparrow (r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}}, R^p_q) \text{ where} \\
r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}} & = & (\looparrowright_\Phi (r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \bigcirc \looparrowright_\Phi (r_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}), \lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}}) \\
R^q_p & = & R_{\mu^p_{R_q}} \cup R_{\mu^q_{R_p}} \cup res(r_{\mu^p_{R_q}}^{-1} r_p) \ \cup \ res(r_{\mu^q_{R_p}}^{-1} r_q)
\end{array}
$$

$$
\begin{array}{lcl}
p' \parallel q' & = & \bigcup_{\Phi'\in\Psi'} \uparrow ((r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}, \lambda_{\mu^{p'}_{R_{q'}}} \cup \lambda_{\mu^{q'}_{R_{p'}}}), R^{p'}_{q'}) \text{ where} \\
r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}} & = & (\looparrowright_{\Phi'} (r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}}) \bigcirc \looparrowright_{\Phi'} (r_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}}), \lambda_{\mu^{p'}_{R_{q'}}} \cup \lambda_{\mu^{q'}_{R_{p'}}}) \\
R^{q'}_{p'} & = & R_{\mu^{p'}_{R_{q'}}} \cup R_{\mu^{q'}_{R_{p'}}} \cup res(r_{\mu^{p'}_{R_{q'}}}^{-1} r_{p'}) \cup res(r_{\mu^{q'}_{R_{p'}}}^{-1} r_{q'})
\end{array}
$$

Proving that $p \parallel q \sqsubseteq p' \parallel q'$ is equivalent to proving that

$\forall \Phi' \in \Psi'.\ \exists \Phi \in \Psi.\ (r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}}, R^p_q) \sqsubseteq (r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}, R^{p'}_{q'})$

Hence, we have to prove that

$\forall \Phi' \in \Psi'.\ \exists \Phi \in \Psi.\ r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}} \preceq r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}$ and

$R_p \cup R_q \cup res(r_{\mu^p_{R_q}}^{-1} r_p) \cup res(r_{\mu^q_{R_p}}^{-1} r_q) \supseteq$
$R_{p'} \cup R_{q'} \cup res(r_{\mu^{p'}_{R_{q'}}}^{-1} r_{p'}) \cup res(r_{\mu^{q'}_{R_{p'}}}^{-1} r_{q'}) \cup res((r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}})^{-1} (r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}))$

First, we have to prove that

$r_{\mu^p_{R_q}} \parallel_\Phi r_{\mu^q_{R_p}} \preceq r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}$

This is equivalent to proving that

$$
\left\{
\begin{array}{l}
\looparrowright_\Phi (r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \ \bigcirc \ \looparrowright_\Phi (r_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}) \rhd \\
\looparrowright_{\Phi'} (r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}}) \bigcirc \looparrowright_{\Phi'} (r_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}}) \\
\lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}} \ \subseteq \ \lambda_{\mu^{p'}_{R_{q'}}} \cup \lambda_{\mu^{q'}_{R_{p'}}}
\end{array}
\right.
$$

Let us prove the first constraint. The proof is done through the following steps:

1. First, it is easy to prove that $r_{\mu^p_{R_q}} \rhd r_{\mu^{p'}_{R_{q'}}}$ and $r_{\mu^q_{R_p}} \rhd r_{\mu^{q'}_{R_{p'}}}$. The proof is similar to the one done in the weak sequential composition monotonicity section (first item).

   Let Φ' be a relation in Ψ'. We define $\Phi \subseteq \Phi'$ such that $\Phi = \Phi'_{|\xi(r_{\mu^p_{R_q}}) \times \xi(r_{\mu^q_{R_p}})}$. It is clear that the transitive closure of Φ is a partial order relation since it is a subset of Φ'. Besides, we define the predecessors of an event e in a relation Φ as follows:

   $predecessors(e, \Phi) \ = \ \{e' \mid (e', e) \in \Phi\}$

   We claim that $\forall e \in \xi(r_{\mu^p_{R_q}}).\ predecessors(e, \Phi) = predecessors(e, \Phi')$.

In fact, we have

$R_{\mu^p_{R_q}} \cap res(r_{\mu^q_{R_p}}) = \emptyset$ and $R_{\mu^q_{R_p}} \cap res(r_{\mu^p_{R_q}}) = \emptyset$

Moreover, we have

$R_{\mu^{p'}_{R_{q'}}} \cap res(r_{\mu^{q'}_{R_{p'}}}) = \emptyset$ and $R_{\mu^{q'}_{R_{p'}}} \cap res(r_{\mu^{p'}_{R_{q'}}}) = \emptyset$

This means that

$res(r^{-1}_{\mu^q_{R_p}} r_{\mu^{q'}_{R_{p'}}}) \cap res(r_{\mu^p_{R_q}}) = \emptyset$ since $res(r^{-1}_{\mu^q_{R_p}} r_{\mu^{q'}_{R_{p'}}}) \subseteq R_{\mu^q_{R_p}}$ and
$res(r^{-1}_{\mu^p_{R_q}} r_{\mu^{p'}_{R_{q'}}}) \cap res(r_{\mu^q_{R_p}}) = \emptyset$ since $res(r^{-1}_{\mu^p_{R_q}} r_{\mu^{p'}_{R_{q'}}}) \subseteq R_{\mu^p_{R_q}}$

Hence, it is clear that

$\forall e \in \xi(r_{\mu^p_{R_q}}).\ \forall e' \in \xi(r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_{\mu^q_{R_p}}).\ (e', e) \notin \Phi'$

This proves the claim that $\forall e \in \xi(r_{\mu^p_{R_q}}).\ predecessors(e, \Phi) = predecessors(e, \Phi')$.

We also have

$\forall e \in \xi(r_{\mu^q_{R_p}}).\ predecessors(e, \Phi) = predecessors(e, \Phi')$

In fact, $\forall e \in \xi(r_{\mu^q_{R_p}}).\ \forall e' \in \xi(r_{\mu^{p'}_{R_{q'}}}) \setminus \xi(r_{\mu^p_{R_q}}).\ (e', e) \notin \Phi'$.

Now, we have to prove that

$\Lsh_{\Phi} (M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \rhd \Lsh_{\Phi'} (M_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}})$ and
$\Lsh_{\Phi} (M_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}) \rhd \Lsh_{\Phi'} (M_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}})$

Let

$$\begin{array}{lcl} Q & = & \{(\Lsh_{\Phi} (M_1, r_2), \Lsh_{\Phi'} (M'_1, r'_2)) \mid M_1 \rhd M'_1 \wedge \pi_1(r_2) \rhd \pi_1(r'_2) \wedge \\ & & \forall b \in \xi(r_1).\ \forall e' \in \xi(r'_2) \setminus \xi(r_2).\ (e', b) \notin \Phi'\} \\ \\ Q' & = & \{(\Lsh_{\Phi} (M_2, r_1), \Lsh_{\Phi'} (M'_2, r'_1)) \mid M_2 \rhd M'_2 \wedge \pi_1(r_1) \rhd \pi_1(r'_1) \wedge \\ & & \forall b \in \xi(r_2).\ \forall e' \in \xi(r'_1) \setminus \xi(r_1).\ (e', b) \notin \Phi'\} \end{array}$$

We have to prove that $Q \subseteq \mathcal{F}(Q)$ and $Q' \subseteq \mathcal{F}(Q')$.

Since $\forall a \in \xi(r_1).\ \forall e' \in \xi(r'_2) \setminus \xi(r_2).\ (e', a) \notin \Phi'$, we have

$\forall b \in dom(M_1).\ \forall e' \in \xi(r'_2) \setminus \xi(r_2).\ (e', \pi_2(b)) \notin \Phi'$

This means that

$\forall b \in dom(M_1).\ predecessors(\pi_2(b), \Phi) \ = \ predecessors(\pi_2(b), \Phi')$

Hence, we have

$\forall b \in dom(M_1).\ \{a \in \xi(r_2) \mid (a, \pi_2(b)) \in \Phi\} = \{a \in \xi(r'_2) \mid (a, \pi_2(b)) \in \Phi'\}$

By the same strategy, we prove that

$\forall b \in dom(M_2).\ \{a \in \xi(r_1) \mid (a, \pi_2(b)) \in \Phi\} = \{a \in \xi(r'_1) \mid (a, \pi_2(b)) \in \Phi'\}$

Moreover, we have

$$\begin{aligned}
dom(\Lsh_{\Phi}(M_1, r_2)) &= \{(\pi_1(b) \cup \{a \in \xi(r_2) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mid b \in dom(M_1)\} \\
dom(\Lsh_{\Phi'}(M'_1, r'_2)) &= \{(\pi_1(b) \cup \{a \in \xi(r'_2) \mid (a, \pi_2(b)) \in \Phi'\}, \pi_2(b)) \mid b \in dom(M'_1)\} \\
dom(\Lsh_{\Phi}(M_2, r_1)) &= \{(\pi_1(b) \cup \{a \in \xi(r_1) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mid b \in dom(M_2)\} \\
dom(\Lsh_{\Phi'}(M'_2, r'_1)) &= \{(\pi_1(b) \cup \{a \in \xi(r'_1) \mid (a, \pi_2(b)) \in \Phi'\}, \pi_2(b)) \mid b \in dom(M'_2)\}
\end{aligned}$$

Since $dom(M_1) \subseteq dom(M'_1)$ and $dom(M_2) \subseteq dom(M'_2)$, it is clear now that
$dom(\Lsh_{\Phi}(M_1, r_2)) \subseteq dom(\Lsh_{\Phi'}(M'_1, r'_2))$ and
$dom(\Lsh_{\Phi}(M_2, r_1)) \subseteq dom(\Lsh_{\Phi'}(M'_2, r'_1))$

The rest of the proof is the same as the one done in the weak sequential composition (first item).

Since we have
$(\Lsh_{\Phi}(M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}), \Lsh_{\Phi'}(M_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}})) \in Q$ and
$(\Lsh_{\Phi}(M_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}), \Lsh_{\Phi'}(M_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}})) \in Q$

We conclude that
$\Lsh_{\Phi}(M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \rhd \Lsh_{\Phi}(M_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}})$ and
$\Lsh_{\Phi}(M_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}) \rhd \Lsh_{\Phi'}(M_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}})$

2. We have to prove that
   $\forall a \in dom(\Lsh_{\Phi}(M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}))$.
   $Successors(a, N_{\Phi}(r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}})) \rhd Successors(a, N_{\Phi'}(r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}}))$

   Let

   $$\begin{aligned}
   Q \;=\; &\{(Successors(a, N_{\Phi}(r_1, r_2)), Successors(a, N_{\Phi'}(r'_1, r'_2))) \mid a \in N_{\Phi}(r_1, r_2) \wedge \\
   &r_1 \preceq r'_1 \wedge r_2 \preceq r'_2 \wedge \Lsh_{\Phi}(\pi_1(r_1), r_2) \rhd \Lsh_{\Phi'}(\pi_1(r'_1), r'_2)\}
   \end{aligned}$$

   We have to prove that $Q \subseteq \mathcal{F}(Q)$.

   Let $(Successors(a, N_{\Phi}(r_1, r_2)), Successors(a, N_{\Phi'}(r'_1, r'_2))) \in Q$. We have

   $$N_{\Phi}(r_1, r_2) \;=\; \varphi(\Lsh_{\Phi}(\pi_1(r_1), r_2)) \;\cup\; \varphi(\Lsh_{\Phi}(\pi_1(r_2), r_1)) \text{ and}$$

   $$N_{\Phi'}(r'_1, r'_2) \;=\; \varphi(\Lsh_{\Phi'}(\pi_1(r'_1), r'_2)) \;\cup\; \varphi(\Lsh_{\Phi'}(\pi_1(r'_2), r'_1))$$

   Since $\Lsh_{\Phi}(\pi_1(r_1), r_2) \rhd \Lsh_{\Phi'}(\pi_1(r'_1), r'_2)$, we can prove easily that
   $\varphi(\Lsh_{\Phi}(\pi_1(r_1), r_2)) \subseteq \varphi(\Lsh_{\Phi'}(\pi_1(r'_1), r'_2))$ and
   $\varphi(\Lsh_{\Phi}(\pi_1(r_2), r_1)) \subseteq \varphi(\Lsh_{\Phi'}(\pi_1(r'_2), r'_1))$

   We conclude that $N_{\Phi}(r_1, r_2) \subseteq N_{\Phi'}(r'_1, r'_2)$.

Now, let $b \in dom(Successors(a, N_\Phi(r_1, r_2)))$. Hence, we have

$b \in N_\Phi(r_1, r_2)$ and $(\pi_2(a), \pi_2(b)) \in \Phi$

Since $N_\Phi(r_1, r_2) \subseteq N_{\Phi'}(r'_1, r'_2)$ and $\Phi \subseteq \Phi'$, we also have

$b \in N_{\Phi'}(r'_1, r'_2)$ and $(\pi_2(a), \pi_2(b)) \in \Phi'$

This means that

$b \in dom(Successors(a, N_{\Phi'}(r'_1, r'_2)))$

Hence, $dom(Successors(a, N_\Phi(r_1, r_2))) \subseteq dom(Successors(a, N_{\Phi'}(r'_1, r'_2)))$.

From the definition of the function $Successors$, it follows that

$$\begin{array}{lcl} Successors(a, N_\Phi(r_1, r_2))(b) & = & Successors(b, N_\Phi(r_1, r_2)) \text{ and} \\ Successors(a, N_{\Phi'}(r'_1, r'_2))(b) & = & Successors(b, N_{\Phi'}(r'_1, r'_2)) \end{array}$$

Therefore, $\forall b \in dom(Successors(a, N_\Phi(r_1, r_2)))$.

$(Successors(a, N_\Phi(r_1, r_2))(b), Successors(a, N_{\Phi'}(r'_1, r'_2))(b)) \in Q$

This means that $(Successors(a, N_\Phi(r_1, r_2)), Successors(a, N_{\Phi'}(r'_1, r'_2))) \in \overline{\mathcal{F}}$.

Since $dom(\Lsh_\Phi (M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}})) \subseteq N_\Phi(r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}})$, we conclude that we have particularly

$\forall a \in dom(\Lsh_\Phi (M_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}))$.
$Successors(a, N_\Phi(r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}})) \rhd Successors(a, N_{\Phi'}(r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}}))$

3. We have to prove that

$\looparrowright_\Phi (r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \rhd \looparrowright_{\Phi'} (r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}})$

$\looparrowright_\Phi (r_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}) \rhd \looparrowright_{\Phi'} (r_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}})$

Let

$$\begin{array}{lcl} Q & = & \{(\looparrowright_\Phi (r_1, r_2), \looparrowright_{\Phi'} (r'_1, r'_2)) \mid \pi_1(r_1) \rhd \pi_1(r'_1) \ \wedge \ \pi_1(r_2) \rhd \pi_1(r'_2)\} \cup \\ & & \{(Successors(a, N_\Phi(r_1, r_2)), Successors(a, N_{\Phi'}(r'_1, r'_2))) \mid a \in N_\Phi(r_1, r_2) \wedge \\ & & \ r_1 \preceq r'_1 \wedge r_2 \preceq r'_2 \wedge \Lsh_\Phi (\pi_1(r_1), r_2) \rhd \Lsh_{\Phi'} (\pi_1(r'_1), r'_2)\} \\ Q' & = & \{(\looparrowright_\Phi (r_2, r_1), \looparrowright_{\Phi'} (r'_2, r'_1)) \mid \pi_1(r_1) \rhd \pi_1(r'_1) \ \wedge \ \pi_1(r_2) \rhd \pi_1(r'_2)\} \cup \\ & & \{(Successors(a, N_\Phi(r_2, r_1)), Successors(a, N_{\Phi'}(r'_2, r'_1))) \mid a \in N_\Phi(r_2, r_1) \wedge \\ & & \ r_1 \preceq r'_1 \wedge r_2 \preceq r'_2 \wedge \Lsh_\Phi (\pi_1(r_2), r_1) \rhd \Lsh_{\Phi'} (\pi_1(r'_2), r'_1)\} \end{array}$$

We have to prove that $Q \subseteq \mathcal{F}(Q)$ and $Q' \subseteq \mathcal{F}(Q')$.

Let us prove that $Q \subseteq \mathcal{F}(Q)$.

Let $(\looparrowright_\Phi (r_1, r_2), \looparrowright_{\Phi'} (r'_1, r'_2)) \in Q$.

Since we have $\Lsh_{\Phi} (\pi_1(r_1), r_2) \rhd \Lsh_{\Phi'} (\pi_1(r'_1), r'_2)$, we get

$dom(\Lsh_{\Phi} (\pi_1(r_1), r_2)) \ \subseteq dom(\Lsh_{\Phi} (\pi_1(r'_1), r'_2))$

This means that $dom(\looparrowright_{\Phi} (r_1, r_2)) \subseteq dom(\looparrowright_{\Phi'} (r'_1, r'_2))$ (see the definition of the operator $\looparrowright$). By the same strategy, we also get $dom(\looparrowright_{\Phi} (r_2, r_1)) \subseteq dom(\looparrowright_{\Phi'} (r'_2, r'_1))$.

Now, we claim that

$\forall a \in dom(\looparrowright_{\Phi} (r_2, r_1)).\ (\looparrowright_{\Phi} (r_2, r_1)(a), \looparrowright_{\Phi'} (r'_2, r'_1)(a)) \in Q$

We use the already done proof stating that

$(Successors(a, N_{\Phi}(r_1, r_2)), Successors(a, N_{\Phi}(r'_1, r'_2))) \in \overline{\mathcal{F}}$

In fact, we have

$\forall a \in dom(\Lsh_{\Phi} (\pi_1(r_1), r_2)).\ \looparrowright_{\Phi} (r_1, r_2)(a) = Successors(a, N_{\Phi}(r_1, r_2))$ and
$\forall a \in dom(\Lsh_{\Phi} (\pi_1(r_1), r_2)).\ \looparrowright_{\Phi'} (r'_1, r'_2)(a) = Successors(a, N_{\Phi}(r'_1, r'_2))$

This proves that $Q \subseteq \mathcal{F}(Q)$.

Proving that $Q' \subseteq \mathcal{F}(Q')$ is similar to proving that $Q \subseteq \mathcal{F}(Q)$.

Hence, it is clear now from all the previous reasoning that we have particularly

$\looparrowright_{\Phi} (r_{\mu^p_{R_q}}, r_{\mu^q_{R_p}}) \rhd \looparrowright_{\Phi'} (r_{\mu^{p'}_{R_{q'}}}, r_{\mu^{q'}_{R_{p'}}})$ and
$\looparrowright_{\Phi} (r_{\mu^q_{R_p}}, r_{\mu^p_{R_q}}) \rhd \looparrowright_{\Phi'} (r_{\mu^{q'}_{R_{p'}}}, r_{\mu^{p'}_{R_{q'}}})$

4. We have to prove that

$\looparrowright_{\Phi} (r_1, r_2) \bigcirc \looparrowright_{\Phi} (r_2, r_1) \rhd \looparrowright_{\Phi'} (r'_1, r'_2) \bigcirc \looparrowright_{\Phi'} (r'_2, r'_1)$

We have

$\looparrowright_{\Phi} (r_1, r_2) \bigcirc \looparrowright_{\Phi} (r_2, r_1) =$
$[a \mapsto \looparrowright_{\Phi} (r_1, r_2)(a) \mid a \in dom(\looparrowright_{\Phi} (r_1, r_2)) \ \wedge \ \pi_1(a) = \emptyset] \dagger$
$[a \mapsto \looparrowright_{\Phi} (r_2, r_1)(a) \mid a \in dom(\looparrowright_{\Phi} (r_2, r_1)) \ \wedge \ \pi_1(a) = \emptyset]$

$\looparrowright_{\Phi'} (r'_1, r'_2) \bigcirc \looparrowright_{\Phi'} (r'_2, r'_1) =$
$[a \mapsto \looparrowright_{\Phi'} (r'_1, r'_2)(a) \mid a \in dom(\looparrowright_{\Phi'} (r'_1, r'_2)) \ \wedge \ \pi_1(a) = \emptyset] \dagger$
$[a \mapsto \looparrowright_{\Phi'} (r'_2, r'_1)(a) \mid a \in dom(\looparrowright_{\Phi'} (r'_2, r'_1)) \ \wedge \ \pi_1(a) = \emptyset]$

Let $a \in dom(\looparrowright_{\Phi} (r_1, r_2))$ such that $\pi_1(a) = \emptyset$, i.e. a is an initial.

We have $a \in dom(\looparrowright_{\Phi'} (r'_1, r'_2))$ since $\looparrowright_{\Phi} (r_1, r_2) \rhd \looparrowright_{\Phi'} (r'_1, r'_2)$.

From this, we deduce that

$[a \mapsto \looparrowright_{\Phi} (r_1, r_2)(a) \mid a \in dom(\looparrowright_{\Phi} (r_1, r_2)) \ \wedge \ \pi_1(a) = \emptyset] \rhd$
$[a \mapsto \looparrowright_{\Phi'} (r'_1, r'_2)(a) \mid a \in dom(\looparrowright_{\Phi'} (r'_1, r'_2)) \wedge \pi_1(a) = \emptyset]$

By the same strategy, we get

$$[a \mapsto \looparrowright_{\Phi} (r_2, r_1)(a) \mid a \in dom(\looparrowright_{\Phi} (r_2, r_1)) \ \wedge\ \pi_1(a) = \emptyset] \rhd$$
$$[a \mapsto \looparrowright_{\Phi'} (r'_2, r'_1)(a) \mid a \in dom(\looparrowright_{\Phi'} (r'_2, r'_1)) \ \wedge\ \pi_1(a) = \emptyset]$$

Since $dom(\pi_1(r_1))$ and $dom(\pi_1(r'_2))$ are disjoint, we can prove easily, from the definition of the operator $\looparrowright$, that $dom(\looparrowright_{\Phi} (r_1, r_2))$ and $dom(\looparrowright_{\Phi'} (r'_2, r'_1))$ are disjoint.

By Lemma 7.8.1, we conclude that

$$\looparrowright_{\Phi} (r_1, r_2) \bigcirc \looparrowright_{\Phi} (r_2, r_1) \rhd \looparrowright_{\Phi'} (r'_1, r'_2) \bigcirc \looparrowright_{\Phi'} (r'_2, r'_1)$$

This means that $r_1 \parallel_{\Phi} r_2 \rhd r'_1 \parallel_{\Phi'} r'_2$.

Proving that $\lambda_{\mu^{p}_{R_q}} \cup \lambda_{\mu^{q}_{R_p}} \subseteq \lambda_{\mu^{p'}_{R_{q'}}} \cup \lambda_{\mu^{q'}_{R_{p'}}}$ is straightforward since $r_{\mu^{p}_{R_q}} \preceq r_{\mu^{p'}_{R_{q'}}}$ and $r_{\mu^{q}_{R_p}} \preceq r_{\mu^{q'}_{R_{p'}}}$.

Hence, we conclude that we have particularly

$$r_{\mu^{p}_{R_q}} \parallel_{\Phi} r_{\mu^{q}_{R_p}} \preceq r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}.$$

For the resource constraint, we have to prove that

$$R_p \cup R_q \cup res(r_{\mu^{p}_{R_q}}^{-1} r_p) \cup res(r_{\mu^{q}_{R_p}}^{-1} r_q) \supseteq$$
$$R_{p'} \cup R_{q'} \cup res(r_{\mu^{p'}_{R_{q'}}}^{-1} r_{p'}) \cup res(r_{\mu^{q'}_{R_{p'}}}^{-1} r_{q'}) \cup$$
$$res((r_{\mu^{p}_{R_q}} \parallel_{\Phi} r_{\mu^{q}_{R_p}})^{-1} (r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}))$$

We have

$$\begin{aligned} \xi(r_{\mu^{p}_{R_q}} \parallel_{\Phi} r_{\mu^{q}_{R_p}}) &= dom(\lambda_{\mu^{p}_{R_q}} \cup \lambda_{\mu^{q}_{R_p}}) \\ &= dom(\lambda_{\mu^{p}_{R_q}}) \cup dom(\lambda_{\mu^{q}_{R_p}}) \\ &= \xi(r_{\mu^{p}_{R_q}}) \cup \xi(r_{\mu^{q}_{R_p}}) \end{aligned}$$

In addition, we have

$$\xi(r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}) = \xi(r_{\mu^{p'}_{R_{q'}}}) \cup \xi(r_{\mu^{q'}_{R_{p'}}})$$

Since $\xi(r_{\mu^{p'}_{R_{q'}}}) \cap \xi(r_{\mu^{q}_{R_p}}) = \emptyset$ and $\xi(r_{\mu^{q'}_{R_{p'}}}) \cap \xi(r_{\mu^{p}_{R_q}}) = \emptyset$, we have

$$\begin{aligned} \xi((r_{\mu^{p}_{R_q}} \parallel_{\Phi} r_{\mu^{q}_{R_p}})^{-1} (r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}})) &= \xi(r_{\mu^{p'}_{R_{q'}}} \parallel_{\Phi'} r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_{\mu^{p}_{R_q}} \parallel_{\Phi} r_{\mu^{q}_{R_p}}) \\ &= (\xi(r_{\mu^{p'}_{R_{q'}}}) \cup \xi(r_{\mu^{q'}_{R_{p'}}})) \setminus (\xi(r_{\mu^{p}_{R_q}}) \cup \xi(r_{\mu^{q}_{R_p}})) \\ &= (\xi(r_{\mu^{p'}_{R_{q'}}}) \setminus \xi(r_{\mu^{p}_{R_q}})) \cup (\xi(r_{\mu^{q'}_{R_{p'}}}) \setminus \xi(r_{\mu^{q}_{R_p}})) \end{aligned}$$

Following the same strategy, we also get

$$(\lambda_{\mu^{p'}_{R_{q'}}} \cup \lambda_{\mu^{q'}_{R_{p'}}}) \setminus (\lambda_{\mu^{p}_{R_q}} \cup \lambda_{\mu^{q}_{R_p}}) \quad = \quad (\lambda_{\mu^{p'}_{R_{q'}}} \setminus \lambda_{\mu^{p}_{R_q}}) \cup (\lambda_{\mu^{q'}_{R_{p'}}} \setminus \lambda_{\mu^{q}_{R_p}})$$

Hence, like what we did for the weak sequential composition and using equation 7.14, we deduce that

$$res((r_{\mu^{p}_{R_q}} \|_{\Phi} r_{\mu^{q}_{R_p}})^{-1}(r_{\mu^{p'}_{R_{q'}}} \|_{\Phi'} r_{\mu^{q'}_{R_{p'}}})) \quad = \quad res({r_{\mu^{p}_{R_q}}}^{-1} r_{\mu^{p'}_{R_{q'}}}) \cup res({r_{\mu^{q}_{R_p}}}^{-1} r_{\mu^{q'}_{R_{p'}}})$$

Since $r_{\mu^{p}_{R_q}} \preceq r_{\mu^{p'}_{R_{q'}}} \preceq r_{p'}$ and $r_{\mu^{q}_{R_p}} \preceq r_{\mu^{q'}_{R_{p'}}} \preceq r_{q'}$ and using equation 7.3, we get

$$res({r_{\mu^{p}_{R_q}}}^{-1} r_{\mu^{p'}_{R_{q'}}}) \cup res({r_{\mu^{p'}_{R_{q'}}}}^{-1} r_{p'}) \quad = \quad res({r_{\mu^{p}_{R_q}}}^{-1} r_{p'})$$

$$res({r_{\mu^{q}_{R_p}}}^{-1} r_{\mu^{q'}_{R_{p'}}}) \cup res({r_{\mu^{q'}_{R_{p'}}}}^{-1} r_{q'}) \quad = \quad res({r_{\mu^{q}_{R_p}}}^{-1} r_{q'})$$

We also have

$$res({r_{\mu^{p}_{R_q}}}^{-1} r_{p'}) \quad = \quad res({r_{\mu^{p}_{R_q}}}^{-1} r_p) \cup res({r_p}^{-1} r_{p'}) \text{ and}$$

$$res({r_{\mu^{q}_{R_p}}}^{-1} r_{q'}) \quad = \quad res({r_{\mu^{q}_{R_p}}}^{-1} r_q) \cup res({r_q}^{-1} r_{q'})$$

Since we have

$$R_p \supseteq R_{p'} \cup res({r_p}^{-1} r_{p'}) \text{ and } R_q \supseteq R_{q'} \cup res({r_q}^{-1} r_{q'})$$

We deduce that

$$R_p \cup R_q \cup res({r_{\mu^{p}_{R_q}}}^{-1} r_p) \cup res({r_{\mu^{q}_{R_p}}}^{-1} r_q) \supseteq$$
$$R_{p'} \cup R_{q'} \cup res({r_p}^{-1} r_{p'}) \cup res({r_{\mu^{p}_{R_q}}}^{-1} r_p) \cup res({r_q}^{-1} r_{q'}) \cup res({r_{\mu^{q}_{R_p}}}^{-1} r_q)$$

Finally, we conclude that

$$R_p \cup R_q \cup res({r_{\mu^{p}_{R_q}}}^{-1} r_p) \cup res({r_{\mu^{q}_{R_p}}}^{-1} r_q) \supseteq$$
$$R_{p'} \cup R_{q'} \cup res({r_{\mu^{p'}_{R_{q'}}}}^{-1} r_{p'}) \cup res({r_{\mu^{q'}_{R_{p'}}}}^{-1} r_{q'}) \cup res((r_{\mu^{p}_{R_q}} \|_{\Phi} r_{\mu^{q}_{R_p}})^{-1}(r_{\mu^{p'}_{R_{q'}}} \|_{\Phi'} r_{\mu^{q'}_{R_{p'}}}))$$

7.8.6 Fixed Point Existence

Let $P \in \mathcal{L}$, $\theta : \zeta \to \mathbb{D}$ and $f : \mathbb{D} \to \mathbb{D}$ be a function defined by

$$f(Y) \quad = \quad [\![P]\!](\theta[X \mapsto Y]) \cap \uparrow (([\,], \emptyset), \mathcal{R})$$

Let E be a subspace of $\mathbb{D}$ and defined as follows:

$$E \quad = \quad \{X \subseteq \mathbb{C} \mid X \neq \emptyset \wedge \exists x \in \mathbb{C}.\ X = \uparrow x\}$$

It is easy to prove that $(E, \sqsubseteq)$ is a cpo where $\sqsubseteq$ is the reverse containment order. The proof uses the fact that $(\mathbb{C}, \sqsubseteq_{\mathbb{C}})$ is a ccpo.

In what follows, we define a recursive semantic function over the space E and which dominates the function f. Let θ' be an environment associating process variables with elements in E. Let $[\![\_]\!]'_{\_} : \mathcal{L} \rightarrow \theta' \rightarrow E$ be a semantic interpretation function that associates each term of the language $\mathcal{L}$ with an element of the space E.

Let $P \in \mathcal{L}$, $\theta' : \zeta \rightarrow E$ and $g : E \rightarrow E$ defined by

$$g(Y) \quad = \quad [\![P]\!]'(\theta'[X \mapsto Y]) \cap \uparrow (([\,], \emptyset), \mathcal{R})$$

The function g is the semantic function used to define the recursion semantics over E (the equivalent of f over $\mathbb{D}$).

In the sequel, we define the semantics of each term of the language $\mathcal{L}$ over the space E and we prove that $\forall Y \in E.\ f(Y) \sqsubseteq g(Y)$ by structural induction on the term P.

For SKIP, STOP, a and X, the semantics of these terms over E are defined in the same way as over $\mathbb{D}$. The proof is straightforward for SKIP, STOP and a. In addition, it is clear that if $P = X$, we have $\forall Y \in E.\ f(Y) = Y \sqsubseteq g(Y) = Y$.

For the case $P = \oplus_{i \in I} P_i$, the induction hypothesis states that

$\forall Y \in E.\ \forall i \in I.\ [\![P_i]\!](\theta[X \mapsto Y]) \sqsubseteq [\![P_i]\!]'(\theta'[X \mapsto Y])$

We define the semantics of the non-deterministic choice in E as follows:

$[\![\oplus_{i \in I} P_i]\!]'(\theta'[X \mapsto Y]) = [\![P_{\chi(I)}]\!]'(\theta'[X \mapsto Y])$ where
$\chi : \mathcal{P}_0(J) \rightarrow I$ is a choice function, J is an index set containing I and
$\mathcal{P}_0(J)$ denotes the set of non-empty subsets of J.

Hence, it is clear in this case that $\forall Y \in E.\ f(Y) \sqsubseteq g(Y)$ by induction hypothesis.

Now, we deal with hiding, strict sequential, weak sequential and parallel composition. Let F^+ be a semantic function operating on the elements of E and defined using F in the same way in which $\tilde{\omega}^+$ is defined using $\tilde{\omega}$ (see Section 7.5). For strict sequential, weak sequential and parallel composition, proving that $\forall Y \in E.\ f(Y) \sqsubseteq g(Y)$, is equivalent to proving that

$$\begin{aligned}\forall Y \in E.\ \tilde{\omega}^+([\![P]\!](\theta[X \mapsto Y]), [\![Q]\!](\theta[X \mapsto Y])) \sqsubseteq \\ F^+([\![P]\!]'(\theta'[X \mapsto Y]), [\![Q]\!]'(\theta'[X \mapsto Y])),\ \text{i.e.}\end{aligned}$$

F^+ dominates $\tilde{\omega}^+$ with respect to the order $\sqsubseteq$.

To prove this, we have only to prove that F dominates $\tilde{\omega}$. In the following, we define a dominating semantic function F for each semantic function $\tilde{\omega}$.

For hiding, the strategy is similar with the consideration of unary semantic functions.

$\tilde{\omega}$	F
$;$	$;$ (the semantic function defined in section 7.5.3)
$\backslash$	$\backslash$ (the semantic function defined in section 7.5.4)
$\circ$	$\circ$ (the semantic function defined in section 7.5.5)
$\parallel$	$F \quad : \quad \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ defined by

$$F(p,q) \quad = \begin{cases} ((M_{\mu^p_{R_q}} \parallel_\Phi M_{\mu^q_{R_p}}, \lambda_{\mu^p_{R_q}} \cup \lambda_{\mu^q_{R_p}}), R^p_q), & \text{if } R_{\mu^p_{R_q}} \cap res(r_{\mu^q_{R_p}}) = \emptyset \wedge \\ & \quad R_{\mu^q_{R_p}} \cap res(r_{\mu^p_{R_q}}) = \emptyset; \\ (([\,], \emptyset), \mathcal{R}), & \text{Otherwise.} \end{cases}$$

Where:

$$R^p_q \quad = \quad R_p \cup R_q \cup res(r_{\mu^p_{R_q}}^{-1} r_p) \cup res(r_{\mu^q_{R_p}}^{-1} r_q) \text{ and}$$

$$\Phi \quad = \quad \{(e,e') \in \xi(r_{\mu^p_{R_q}}) \times \xi(r_{\mu^q_{R_p}}) \mid res(\lambda_{\mu^p_{R_q}}(e)) \cap res(\lambda_{\mu^q_{R_p}}(e')) \neq \emptyset\} \cup \mathcal{D}(\varphi(M_{\mu^p_{R_q}})) \cup \mathcal{D}(\varphi(M_{\mu^q_{R_p}}))$$

For the strict sequential composition, hiding and weak sequential composition, it is straightforward to prove that if F dominates $\tilde{\omega}$, then F^+ dominates $\tilde{\omega}^+$. Moreover, for the parallel composition F^+ is more deterministic than $\tilde{\omega}^+$. Hence, since F^+ dominates $\tilde{\omega}^+$, we deduce that g dominates f.

Moreover, we claim that all the functions F are monotone. In fact, the proofs are already done for some operators (like $;$, $\backslash$ and $\circ$) while the proof of the monotonicity of the dominating function F for the parallel composition is similar to what we did in section 7.8.5. This means that F^+ is also monotone and consequently g is monotone.

Let $Z = \uparrow (([\,], \emptyset), \emptyset)$ and $X_0 = \uparrow (([\,], \emptyset), res(f(Z)))$. We claim that

$\{f^\alpha(X_0) \mid \alpha \in Ord\}$ is directed and that it has a least upper bound.

To prove this claim, we have to prove that X_0 is a postfixed point of g, i.e. $X_0 \sqsubseteq g(X_0)$ and also of f. After that, we use the fact that g dominates f and the proposition 7.5.1 to conclude that $\{f^\alpha(X_0) \mid \alpha \in Ord\}$ has a least upper bound.

Hereafter, we prove that $X_0 \sqsubseteq g(X_0)$. The proof is done by structural induction on P.

- Case $P =$ `SKIP`. In this case, we have

  $X_0 = \uparrow (([\,], \emptyset), \emptyset)$ and $g(X_0) = \uparrow (([\,], \emptyset), \emptyset)$

  We deduce that $X_0 \sqsubseteq g(X_0)$ since $\sqsubseteq$ is reflexive.

- Case $P = \texttt{STOP}$.

  The proof is similar to the case $P = \texttt{SKIP}$ with $X_0 = g(X_0) = \uparrow(([\,],\emptyset),\mathcal{R})$.

- Case $P = a$. In this case, we have

  $X_0 = \uparrow(([\,],\emptyset), res(a))$ and $g(X_0) = \{((([(\emptyset, a_1) \mapsto [\,]], \{(a_1, a)\}), \emptyset)\}$

  Since $(([\,],\emptyset), res(a)) \sqsubseteq (([(\emptyset, a_1) \mapsto [\,]], \{(a_1, a)\}), \emptyset)$, we deduce that $X_0 \sqsubseteq g(X_0)$.

- Case $P = X$. In this case, $g(X_0) = X_0$ and we deduce that $X_0 \sqsubseteq g(X_0)$.

- Case $P = P\backslash R$.

  The induction hypothesis states that

  $X_0 \sqsubseteq [\![P]\!](\theta[X \mapsto X_0])$

  Moreover, we can easily prove that $X_0 \sqsubseteq X_0\backslash R$ and since the semantic function $\backslash$ is monotone, we deduce that

  $X_0 \sqsubseteq X_0\backslash R \sqsubseteq [\![P]\!](\theta[X \mapsto X_0])\backslash R$

  Since $g(X_0) = [\![P]\!](\theta[X \mapsto X_0])\backslash R$, we conlcude that $X_0 \sqsubseteq g(X_0)$.

- Case $P = \oplus_{i\in I} P_i$.

  The induction hypothesis states that $\forall i \in I.\ X_0 \sqsubseteq [\![P_i]\!](\theta[X \mapsto X_0])$.

  We suppose that $\chi(I) = j_0$, then $g(X_0) = [\![P_{j_0}]\!](\theta[X \mapsto X_0])$. Using the induction hypothesis, we deduce that $X_0 \sqsubseteq g(X_0)$.

- $P = P_1\ ;\ P_2$. We have $g(X_0) = [\![P_1\ ;\ P_2]\!](\theta[X \mapsto X_0])$.

  We have to prove that $X_0 \sqsubseteq [\![P_1\ ;\ P_2]\!](\theta[X \mapsto X_0])$.

  The induction hypothesis states that

  $X_0 \sqsubseteq [\![P_1]\!](\theta[X \mapsto X_0])$ and $X_0 \sqsubseteq [\![P_2]\!](\theta[X \mapsto X_0])$

  By compositionality of the semantics, we have

  $[\![P_1\ ;\ P_2]\!](\theta[X \mapsto X_0]) = [\![P_1]\!](\theta[X \mapsto X_0])\ ;\ [\![P_2]\!](\theta[X \mapsto X_0])$

  We can prove easily that $X_0\ ;\ X_0 = X_0$ and by the monotonicity of the strict sequential composition, we deduce that

  $X_0 = X_0\ ;\ X_0 \sqsubseteq g(X_0) = [\![P_1]\!](\theta[X \mapsto X_0])\ ;\ [\![P_2]\!](\theta[X \mapsto X_0])$

- The cases $P = P_1 \circ P_2$ and $P = P_1 \parallel P_2$ are handled in a similar way as the previous case.

Using the fact that g is monotone, we deduce that the set $\{g^{\alpha}(X_0) \mid \alpha \in Ord\}$ is directed. Since E is a cpo, we conclude that $\bigsqcup_{\alpha\in Ord} g^{\alpha}(X_0)$ exists.

Moreover, we claim that X_0 is a postfixed point of f. The proof of this claim is similar to the one done for g.

Now, since g dominates f and X_0 is a postfixed point of f and g and by proposition 7.5.1, we deduce that $\bigsqcup_{\alpha \in Ord} f^{\alpha}(X_0)$ exists. Moreover, since the set $\{f^{\alpha}(X_0) \mid \alpha \in Ord\}$ is indexed over all α and bounded by $\bigsqcup_{\alpha \in Ord} g^{\alpha}(X_0)$, we deduce that $\exists \lambda \in Ord$ such that $f^{\lambda}(X_0) = f^{\lambda+1}(X_0)$. This means that $\bigsqcup_{\alpha \in Ord} f^{\alpha}(X_0)$ is a fixed point of f.

7.9 Healthiness Conditions

Hereafter, we prove that the output of each semantic function satisfies the two healthiness conditions of our model.

7.9.1 First Healthiness Condition Verification

The first healthiness condition states that for each process p, the reflexive transitive closure of $\mathcal{D}(\varphi(M_p))$ is a partial order relation.

First, we provide the following results:

$\forall M, M' \in \mathbb{M}$ such that $dom(M) \cap dom(M') = \emptyset$, we have

$\varphi(M \dagger M') = \varphi(M) \cup \varphi(M')$

We also have $\forall A, B \in \mathcal{P}(\mathcal{P}_f(V) \times V).\ \mathcal{D}(A \cup B) = \mathcal{D}(A) \cup \mathcal{D}(B)$

The proofs of these results are obvious. Moreover, it is simple to prove that $\mathcal{D}$ is monotone with respect to $\subseteq$.

The aforementioned two results and the monotonicity of $\mathcal{D}$ are needed for verifying the first healthiness condition. The proofs related to the first healthiness condition use a co-induction principle defined as follows:

Let $\mathcal{U}'$ be the set of all (S, S') such that $S, S' \in \mathcal{P}(\mathcal{P}_f(V) \times V)$.

Let

$$\begin{array}{lcl} \mathcal{H} & : & \mathcal{P}(\mathcal{U}') \rightarrow \mathcal{P}(\mathcal{U}') \\ \mathcal{H}(Q) & = & \{(\varphi(M), X) \mid M \in \mathbb{M} \wedge dom(M) \subseteq X \wedge \forall a \in dom(M).\ (\varphi(M(a)), X) \in Q\} \end{array}$$

It is easy to show that $\mathcal{H}$ is monotone. Since $\mathcal{U}'$ is a cpo, $\mathcal{H}$ has a greatest fixed point. In what follows, we omit the details about the co-induction proofs in order to make the presentation clear. The proofs are similar to those we did for the monotonicity of the semantic functions.

- For strict sequential composition:

  We have $\mathcal{D}(\varphi(M_{p;q})) = \mathcal{D}(\varphi(M_p))$ if $R_p \neq \emptyset$.

  In this case, the reflexive transitive closure of $\mathcal{D}(\varphi(M_{p;q}))$ is a partial order relation since this is true for $\mathcal{D}(\varphi(M_p))$.

  Otherwise, we have

  $$\begin{array}{lcl} \varphi(M_{p;q}) & = & \varphi(S(M_p, M_q)) \\ & = & \varphi([a \mapsto S(M_p(a), M_q) \mid a \in dom(M_p) \wedge M_p(a) \neq [\,]] \dagger \\ & & [a \mapsto M_q\sigma_{M_p} \mid a \in dom(M_p) \wedge M_p(a) = [\,]]) \end{array}$$

  Let $Q = \{(\varphi(S(M'', M')), \varphi(M) \cup \varphi(M'\sigma_M)) \mid \varphi(M'') \subseteq \varphi(M)\}$.

  We have $(\varphi(S(M_p, M_q)), \varphi(M_p) \cup \varphi(M_q\sigma_{M_p})) \in Q$. By co-induction, we get

  $$\varphi(M_{p;q}) \subseteq \varphi(M_p) \cup \varphi(M_q\sigma_{M_p})$$

  We also have

  $$\begin{array}{lcl} \varphi(M_q\sigma_{M_p}) & = & \bigcup_{a \in dom(M_q)} \varphi(M_q(a)) \cup \\ & & \\ & & \{(Terminals(M_p) \cup \pi_1(a), \pi_2(a)) \mid a \in dom(M_q)\} \end{array}$$

  Hence, we get

  $$\begin{array}{lcl} \mathcal{D}(\varphi(M_{p;q})) & \subseteq & \mathcal{D}(\varphi(M_p)) \cup \mathcal{D}(\varphi(M_q)) \cup \\ & & \\ & & \{(e, \pi_2(b)) \mid e \in Terminals(M_p) \wedge b \in dom(M_q)\} \end{array}$$

  It is clear that the reflexive transitive closure of $\mathcal{D}(\varphi(M_{p;q}))$ is a partial order relation since this property is satisfied by $\mathcal{D}(\varphi(M_p))$ and $\mathcal{D}(\varphi(M_q))$ and their union also satisfies this property since $\xi(r_p)$ and $\xi(r_q)$ are disjoint. Moreover, the relation $\{(e, \pi_2(b)) \mid e \in Terminals(M_p) \wedge b \in dom(M_q)\}$ introduces dependencies only from $\xi(r_p)$ to $\xi(r_q)$.

- For hiding:

  Let R be a finite set of resources. We have

  $$\varphi(H_R(M_p)) = \varphi([(\sigma^+(\pi_1(a)), \sigma(\pi_2(a))) \mapsto H_R(M_p(a)) \mid a \in dom(M_p)])$$

  Let

  $$Q = \{(\varphi(H_R(M)), \sigma'^+(\varphi(M'))) \mid \varphi(M) \subseteq \varphi(M')\} \text{ where}$$

  $$\begin{array}{lcl} \sigma' & : & \mathcal{P}_f(V) \times V \to \mathcal{P}_f(V) \times V \text{ defined by} \\ \sigma'(a) & = & (\sigma^+(\pi_1(a)), \sigma(\pi_2(a))) \end{array}$$

  Note that σ is the substitution defined in Section 7.5.4.

  We have $(\varphi(H_R(M_p)), \sigma'^+(\varphi(M_p))) \in Q$. Using co-induction, we get

  $$\varphi(H_R(M_p)) \subseteq \sigma'^+(\varphi(M_p))$$

From this, we get

$$\begin{array}{lcl}\mathcal{D}(\varphi(H_R(M_p))) & \subseteq & \sigma''(\mathcal{D}(\varphi(M_p))) \text{ where} \\ \sigma''(\mathcal{D}(\varphi(M_p))) & = & \{(\sigma(a), \sigma(b)) \mid (a,b) \in \mathcal{D}(\varphi(M_p))\}\end{array}$$

Since σ is injective and the reflexive transitive closure of $\mathcal{D}(\varphi(M_p))$ is a partial order relation, we deduce that the reflexive transitive closure of $\sigma''(\mathcal{D}(\varphi(M_p)))$ is a partial order relation too.

Finally, we conclude that the reflexive transitive closure of $\mathcal{D}(\varphi(H_R(M_p)))$ is a partial order relation since this property is satisfied by $\sigma''(\mathcal{D}(\varphi(M_p)))$.

- For weak sequential composition:

  Let $\Phi = \{(e, e') \in \xi(r_p) \times \xi(r_q) \mid res(\lambda_p(e)) \cap res(\lambda_q(e')) \neq \emptyset\}$. We claim that

  $$\mathcal{D}(\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \ddagger \Lsh_\Phi (M_q, r_p))) \quad \subseteq \quad \mathcal{D}(\varphi(M_p)) \cup \mathcal{D}(\varphi(M_q)) \cup \Phi$$

  In fact, we have

  $$\begin{array}{l}\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \ddagger \Lsh_\Phi (M_q, r_p)) \qquad = \\ \\ \varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \dagger \\ {[a \mapsto \Lsh_\Phi (M_q, r_p)(a) \mid a \in dom(\Lsh_\Phi (M_q, r_p)) \wedge \pi_1(a) = \emptyset]}) \qquad = \\ \\ \varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p))) \cup \\ \varphi([a \mapsto \Lsh_\Phi (M_q, r_p)(a) \mid a \in dom(\Lsh_\Phi (M_q, r_p)) \wedge \pi_1(a) = \emptyset])\end{array}$$

  We also have

  $$\begin{array}{lcl}\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p))) & = & \bigcup_{a \in dom(M_p)} \{a\} \cup \\ \\ & & \varphi(Succ(a, \Lsh_\Phi (M_q, r_p), \Phi) \dagger \Rsh_\Phi (M_p(a), \Lsh_\Phi (M_q, r_p)))\end{array}$$

  Moreover, from the definition of the function $Succ$, it follows that

  $$\varphi(Succ(a, \Lsh_\Phi (M_q, r_p), \Phi)) \subseteq \varphi(\Lsh_\Phi (M_q, r_p))$$

  Let

  $$\begin{array}{lcl}Q & = & \{(\varphi(\Rsh_\Phi (M, \Lsh_\Phi (M', r))), \varphi(M) \cup \varphi(\Lsh_\Phi (M', r))) \mid \varphi(M) \subseteq \varphi(\pi_1(r))\} \cup \\ \\ & & \{(\varphi(Succ(a, \Lsh_\Phi (M', r), \Phi) \dagger \Rsh_\Phi (M(a), \Lsh_\Phi (M', r))), \varphi(\pi_1(r)) \cup \varphi(\Lsh_\Phi (M', r))) \mid \\ & & \quad a \in dom(M) \wedge \varphi(M) \subseteq \varphi(\pi_1(r))\}\end{array}$$

  We have

  $$(\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p))), \varphi(M_p) \cup \varphi(\Lsh_\Phi (M_q, r_p))) \in Q \text{ and}$$

  $$\begin{array}{l}\forall a \in dom(M_p). \\ (\varphi(Succ(a, \Lsh_\Phi (M_q, r_p), \Phi) \dagger \Rsh_\Phi (M_p(a), \Lsh_\Phi (M_q, r_p))), \varphi(M_p) \cup \varphi(\Lsh_\Phi (M_q, r_p))) \in Q\end{array}$$

  By co-induction, we get

  $$\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p))) \quad \subseteq \quad \varphi(M_p) \cup \varphi(\Lsh_\Phi (M_q, r_p))$$

  We also have

$$\varphi([a \mapsto \Lsh_\Phi (M_q, r_p)(a) \mid a \in dom(\Lsh_\Phi (M_q, r_p)) \mid \pi_1(a) = \emptyset]) \quad \subseteq \quad \varphi(\Lsh_\Phi (M_q, r_p))$$

Hence, we get

$$\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \ddagger \Lsh_\Phi (M_q, r_p)) \quad \subseteq \quad \varphi(M_p) \cup \varphi(\Lsh_\Phi (M_q, r_p))$$

Moreover, we have

$$\varphi(\Lsh_\Phi (M_q, r_p)) \quad = \quad \{(\pi_1(b) \cup \{a \in \xi(r_p) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mid b \in \varphi(M_q)\}$$

Hence, we deduce that

$$\begin{array}{lcl} \mathcal{D}(\varphi(\Lsh_\Phi (M_q, r_p))) & = & \mathcal{D}(\varphi(M_q)) \cup \Phi \\ \mathcal{D}(\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)))) & \subseteq & \mathcal{D}(\varphi(M_p)) \cup \mathcal{D}(\varphi(\Lsh_\Phi (M_q, r_p))) \end{array}$$

This proves the claim that

$$\mathcal{D}(\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \ddagger \Lsh_\Phi (M_q, r_p))) \subseteq \mathcal{D}(\varphi(M_p)) \cup \mathcal{D}(\varphi(M_q)) \cup \Phi$$

The reflexive transitive closure of $\mathcal{D}(\varphi(\Rsh_\Phi (M_p, \Lsh_\Phi (M_q, r_p)) \ddagger \Lsh_\Phi (M_q, r_p)))$ is a partial order relation since this property is satisfied by $\mathcal{D}(\varphi(M_p))$ and $\mathcal{D}(\varphi(M_q))$ and their union also satisfies this property since $\xi(r_p)$ and $\xi(r_q)$ are disjoint. In addition, the relation $\Phi = \{(e, e') \in \xi(r_p) \times \xi(r_q) \mid res(\lambda_p(e)) \cap res(\lambda_q(e')) \neq \emptyset\}$ introduces dependencies only from $\xi(r_p)$ to $\xi(r_q)$.

- For parallel composition:

  We claim that $\mathcal{D}(\varphi(\looparrowright_\Phi (r_p, r_q) \bigcirc \looparrowright_\Phi (r_q, r_p))) \subseteq \Phi$.

  In fact, we have

  $$\varphi(\looparrowright_\Phi (r_p, r_q) \bigcirc \looparrowright_\Phi (r_q, r_p)) =$$

  $$\varphi([a \mapsto \looparrowright_\Phi (r_p, r_q)(a) \mid a \in dom(\looparrowright_\Phi (r_p, r_q)) \wedge \pi_1(a) = \emptyset]) \cup$$

  $$\varphi([a \mapsto \looparrowright_\Phi (r_q, r_p)(a) \mid a \in dom(\looparrowright_\Phi (r_q, r_p)) \wedge \pi_1(a) = \emptyset])$$

  From this equation, we deduce that

  $$\varphi(\looparrowright_\Phi (r_p, r_q) \bigcirc \looparrowright_\Phi (r_q, r_p)) \quad \subseteq \quad \varphi(\looparrowright_\Phi (r_p, r_q)) \cup \varphi(\looparrowright_\Phi (r_q, r_p))$$

  Let

  $$\begin{array}{lcl} Q & = & \{(\varphi(\looparrowright_\Phi (r, r')), N_\Phi(r, r')) \mid r \preceq r'\} \cup \\ & & \{(Successors(a, N_\Phi(r, r')), N_\Phi(r, r')) \mid a \in N_\Phi(r, r') \wedge r \preceq r'\} \cup \\ & & \{(\varphi(\looparrowright_\Phi (r', r)), N_\Phi(r', r)) \mid r \preceq r'\} \cup \\ & & \{(Successors(a, N_\Phi(r', r)), N_\Phi(r', r)) \mid a \in N_\Phi(r', r) \wedge r \preceq r'\} \end{array}$$

  We have

  $(\varphi(\looparrowright_\Phi (r_p, r_q)), N_\Phi(r_p, r_q)) \in Q$ and
  $\forall a \in N_\Phi(r_p, r_q).(Successors(a, N_\Phi(r_p, r_q)), N_\Phi(r_p, r_q)) \in Q$

Also, we have

$(\varphi(\looparrowright_{\Phi} (r_q, r_p)), N_{\Phi}(r_q, r_p)) \ \in \ Q$ and
$\forall a \in N_{\Phi}(r_q, r_p).(Successors(a, N_{\Phi}(r_q, r_p)), N_{\Phi}(r_q, r_p)) \ \in \ Q$

Using co-induction, we get

$$\begin{aligned} \varphi(\looparrowright_{\Phi} (r_p, r_q)) &\subseteq N_{\Phi}(r_p, r_q) \\ \varphi(\looparrowright_{\Phi} (r_q, r_p)) &\subseteq N_{\Phi}(r_q, r_p) = N_{\Phi}(r_p, r_q) \end{aligned}$$

Hence, we deduce that

$$\varphi(\looparrowright_{\Phi} (r_p, r_q) \bigcirc \looparrowright_{\Phi} (r_q, r_p)) \ \subseteq \ N_{\Phi}(r_p, r_q)$$

Since $N_{\Phi}(r_p, r_q) = \varphi(\Lsh_{\Phi} (M_p, r_q)) \cup \varphi(\Lsh_{\Phi} (M_q, r_p))$, we get

$$\varphi(\looparrowright_{\Phi} (r_p, r_q) \bigcirc \looparrowright_{\Phi} (r_q, r_p)) \ \subseteq \ \varphi(\Lsh_{\Phi} (M_p, r_q)) \cup \varphi(\Lsh_{\Phi} (M_q, r_p))$$

This means that

$$\mathcal{D}(\varphi(\looparrowright_{\Phi} (r_p, r_q) \bigcirc \looparrowright_{\Phi} (r_q, r_p))) \ \subseteq \ \mathcal{D}(\varphi(\Lsh_{\Phi} (M_p, r_q))) \cup \mathcal{D}(\varphi(\Lsh_{\Phi} (M_q, r_p)))$$

We also have

$$\begin{aligned} \varphi(\Lsh_{\Phi} (M_p, r_q)) &= \{(\pi_1(b) \cup \{a \in \xi(r_q) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mid b \in \varphi(M_p)\} \\ \varphi(\Lsh_{\Phi} (M_q, r_p)) &= \{(\pi_1(b) \cup \{a \in \xi(r_p) \mid (a, \pi_2(b)) \in \Phi\}, \pi_2(b)) \mid b \in \varphi(M_q)\} \end{aligned}$$

$$\begin{aligned} \mathcal{D}(\varphi(\Lsh_{\Phi} (M_p, r_q))) \cup \mathcal{D}(\varphi(\Lsh_{\Phi} (M_q, r_p))) &= \mathcal{D}(\varphi(M_p)) \cup \mathcal{D}(\varphi(M_q)) \cup \\ &\quad \{(e, e') \in \xi(r_p) \times \xi(r_q) \mid (e, e') \in \Phi\} \cup \\ &\quad \{(e, e') \in \xi(r_q) \times \xi(r_p) \mid (e, e') \in \Phi\} \\ &= \Phi \end{aligned}$$

Hence, we conclude that

$$\mathcal{D}(\varphi(\looparrowright_{\Phi} (r_p, r_q) \bigcirc \looparrowright_{\Phi} (r_q, r_p))) \ \subseteq \ \Phi$$

In the definition of the operator $\|_{\Phi}$, the reflexive transitive closure of Φ is a partial order relation. This also means that the reflexive transitive closure of the relation $\mathcal{D}(\varphi(\looparrowright_{\Phi} (r_p, r_q) \bigcirc \looparrowright_{\Phi} (r_q, r_p)))$ is a partial order relation.

7.9.2 Second Healthiness Condition Verification

We have to prove that the deterministic processes x, which are output of our semantic functions, satisfy the condition $resinf(r_x) \subseteq R_x$. To make the proof simple, we provide an equivalent definition to the function $resinf$. First, we define, the set of actions occurring infinitely often in a process x by the function $alphinf(\lambda_x)$. Hence, we have the following equation:

$$(7.17) \quad \forall x \in \mathbb{C}.\ resinf(r_x) \ = \ res(alphinf(\lambda_x))$$

Moreover, we have:

$$alphinf(\lambda \cup \lambda') \;=\; alphinf(\lambda) \cup alphinf(\lambda')$$

Besides, it is easy to prove that

$$(7.18) \quad \forall A, B \in \mathcal{P}(\Sigma).\ res(A \cup B) \;=\; res(A) \cup res(B)$$

Now, we provide the proofs related to the second healthiness condition. These proofs are established using equations 7.17 and 7.18.

- For strict sequential composition, we have

  $$resinf(r_{p;q}) \;=\; resinf(r_p) \;\subseteq\; R_{p;q} = R_p \text{ if } R_p \neq \emptyset$$

  Otherwise, we have

  $$\begin{aligned} resinf(r_{p;q}) &= res(alphinf(\lambda_{p;q})) \\ &= res(alphinf(\lambda_p \,\cup\, \lambda_q)) \\ &= res(alphinf(\lambda_p) \,\cup\, alphinf(\lambda_q)) \\ &= resinf(r_p) \,\cup\, resinf(r_q) \\ &\subseteq R_p \,\cup\, R_q \;=\; R_q \;=\; R_{p;q} \end{aligned}$$

- For hiding, let R be a finite set of resources and

  $$\begin{aligned} \sigma' &: \Sigma \rightarrow \Sigma \text{ defined by} \\ \sigma'(a) &= \begin{cases} \eta(a), & \text{if } res(a) \subseteq R; \\ a, & \text{Otherwise.} \end{cases} \end{aligned}$$

  Note that η is the injective substitution defined in Section 7.5.4.

  By the same strategy as that followed to prove the monotonicity of the hiding operator, we can prove that $alphinf(\lambda_{p\backslash R}) \;=\; \sigma'^{+}(alphinf(\lambda_p))$.

  Knowing that $\forall a \in \Sigma.\ res(a) = res(\eta(a))$, we have

  $$\begin{aligned} resinf(r_{p\backslash R}) &= res(alphinf(\lambda_{p\backslash R})) \\ &= res(\sigma'^{+}(alphinf(\lambda_p))) \\ &= res(alphinf(\lambda_p)) \\ &= resinf(r_p) \\ &\subseteq R_p \;=\; R_{p\backslash R} \end{aligned}$$

- For weak sequential composition, we use the same strategy as the strict sequential composition and we get

  $$\begin{aligned} resinf(r_{p\circ q}) &= res(alphinf(\lambda_p \,\cup\, \lambda_{\mu^q_{R_p}})) \\ &= res(alphinf(\lambda_p)) \,\cup\, res(alphinf(\lambda_{\mu^q_{R_p}})) \\ &= resinf(r_p) \,\cup\, resinf(r_{\mu^q_{R_p}}) \\ &\subseteq R_p \,\cup\, R_{\mu^q_{R_p}} \;=\; R_p \,\cup\, R_q \,\cup\, res(r^{-1}_{\mu^q_{R_p}}\, r_q) \end{aligned}$$

- For parallel composition, like the strict and weak sequential composition, we have

$$\begin{aligned} resinf(r_{\mu^p_{R_q}} \parallel_{\Phi_i} r_{\mu^q_{R_p}}) &= resinf(r_{\mu^p_{R_q}}) \cup resinf(r_{\mu^q_{R_p}}) \\ &\subseteq R_{\mu^p_{R_q}} \cup R_{\mu^q_{R_p}} = R^p_q \end{aligned}$$

7.10 Conclusion

In this chapter, we presented a new semantic model for true concurrency with unbounded non-determinism. The model is denotational and rests on an extension of the resource pomsets semantics of Gastin and Mislove. We presented the construction of the process space and exhibited its algebraic properties. Moreover, we provided the semantic interpretation of some useful concurrency operators. This led to a fixpoint semantics of recursion. Currently, an accommodation of this model for JVML/CLDC is carried out [66]. Once this extension is performed, we will be able to provide optimization correctness proofs of JVML/CLDC programs.

Chapter 8

Towards A Framework for Validating Optimizations of JVML/CLDC Programs

8.1 Introduction

In the previous chapter, we presented a semantic model for true concurrency with unbounded non-determinism. As mentioned before, the intent of this work is to prepare for the development of a framework in which we can assess the correctness of the optimizations that can be performed on JVML/CLDC programs. To establish such correctness, it is unpractical to do proofs by hand. Moreover, such strategy can be error-prone. For instance, Don Syme [127] discovered some errors when he tried to embed a Java type system [43] in a theorem prover called *Declare*. Accordingly, proof assistants are strongly recommended.

There are several proof assistants such as Isabelle [100], COQ [129], PVS [98]. These theorem provers have been used for the automation of proofs about system verification and validation. It is worth to note that Isabelle is the most adopted one for embedding the semantics of the Java and JVML languages. This statement is corroborated by the publication of several research initiatives about the embedding of the Java language, compilers and virtual machines in this theorem prover [94, 97, 123].

8.2 Related Work

In the sequel, we provide the related work about embedding Java/JVML in Isabelle and the research initiatives about proving optimization correctness and validating compilers.

Nipkow et al. [97] formalized a subset of Java in Isabelle/HOL[1]. The primary objective of their work is to prove the type safety for this subset. In another project, Nipkow et al. [94] formalized a subset of JVML in Isabelle/HOL. The main goal of this formalization is to prove that the verifier is sound and also that the Java compiler is correct. Their strategy is based on an operational semantic style.

Strecker [123] proved the correctness of a Java compiler in an operational small step style. The compiler translates Micro-Java source code[2] into Micro-JVM bytecode. The correctness of the compiler is defined as a commuting diagram (See Figure 6.1 of Chapter 6). The commutation is as follows: suppose that the execution of a statement c transforms a Micro-Java state s into a state s', then for any Micro-JVM state s_1 corresponding to s, the execution of the bytecode resulting from the translation of c yields a state s'_1 corresponding to s'. The designed framework is based on an operational semantic style.

Glesner et al. [14] formalized the generation of code from Static Single Assignment (SSA) form [3] in Isabelle/HOL. They show the correctness of this generation process. The correctness proofs give also checkable correctness criteria characterizing correct compilation results obtained from different implementations of code generation algorithms. This work is also based on an operational semantic style.

Siveroni [117] investigated in his thesis the correctness of optimizations that can be performed on programs written in a functional language. He proved, in an operational big step style, the semantic equivalence between a source program and its optimized version for some basic optimizations (useless variable and expression removal). In a big step style, an optimization is valid if the original and optimized programs return the same value.

Jones et al. [68] proved the optimization correctness for programs written in a simple imperative language using a temporal logic. Each optimization is considered as a rewrite rule guarded by a formula defined in a temporal logic. The correctness proof is reduced to finding an equivalence relation between the source program and the optimized one.

Chambers et al. [70] designed a flow-based optimization framework to prove optimization correctness. The framework consists of defining a language in which optimization can be written, deduce the conditions that should hold in the program in order to perform a specific optimization and verify the semantic equivalence between the source and optimized program.

Our strategy, as discussed before, is denotational. The motivations and justifications

[1]Isabelle/HOL [95] is the specialization of Isabelle for HOL, which is a Higher Order Logic

[2]A small subset of Java

[3]In such form, a variable is assigned just once in the program

behind this choice were presented in previous chapters.

8.3 Isabelle at a Glance

In what follows, we provide a quick overview of the assistant proof in which we did the embedding of the main features of our semantic model.

Isabelle is a generic and interactive theorem prover that provides a logical framework in which theorems about programming languages and programs can be established. Isabelle/HOL is the specialization of Isabelle for HOL. It is written in the Standard ML language (SML). In the sequel, we present some of Isabelle/HOL notations in order to make the understanding of the embedding of the main elements of our semantic model very clear.

8.3.1 Theories

A theory file is a collection of types, functions and theorems. The definition of a theory file is done through the following declaration:

theory $T = B_1 + B_2 \ldots + B_n$:
declarations, definitions and proofs
end

This declaration means that a theory file T is defined. It has B_1, B_2, ... and B_n as direct parents. Hence, all the declarations in these theory files can be used inside T.

8.3.2 Datatypes

A datatype is declared as follows:

datatype$(\alpha_1, \ldots, \alpha_n)$ $t = C_1\ \tau_{11} \ldots \tau_{1k_1} \mid \ldots \mid C_m\ \tau_{m1} \ldots \tau_{mk_m}$

Note that α_i are distinct type variables, C_i are distinct constructor names and τ_{ij} are types. The following example outlines a definition of a list as a datatype:

datatype *'a list = Nil | Cons 'a "'a list"*

8.3.3 Function Declaration

In Isabelle/HOL, total functions are specified using the symbol $\Rightarrow$. A function f: $\alpha \Rightarrow \beta$ takes a parameter whose type is α and returns a result whose type is β. Partial functions are specified using the keyword *option*. More precisely, a partial function f has the signature: $\alpha \Rightarrow \beta$ *option*, where α and β are type variables and β *option* = *None* | *Some* β.

A recursive function is specified using the keyword **primrec** or **recdef**, which is more general than **primrec**. The keyword **primrec** indicates that each recursive call strips off a datatype constructor from one of the arguments. The keyword **recdef** means that recursion does not necessarily involve datatypes.

8.3.4 Tactics

Isabelle comes with many built-in tactics. A tactic is a specific strategy intended to make proofs. The more powerful tactic in Isabelle is *auto*. This tactic tries to apply all the simplification rules that already exist in this proof assistant.

A simplification rule is an equation that allows to rewrite terms. For instance, the term "xs @ [ ]", which is a concatenation between a list "xs" and the empty list, can be simplified into "xs". The corresponding simplification rule is: "xs @ [ ] = xs". To simplify all the subgoals in a proof, the user can use the tactic *simp_all*.

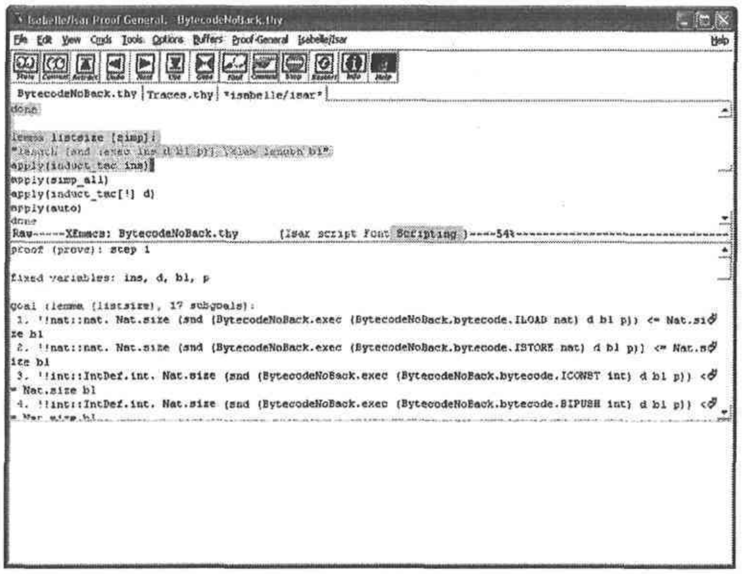
Induction is another interesting tactic in Isabelle. This tactic is called *induct_tac*. In a proof by induction, the user should specify the variable on which the induction is performed.

The use of a tactic in Isabelle is done as follows: **apply**(*tacticname*). To get a full description of Isabelle/HOL, the reader is encouraged to see an interesting tutorial about Isabelle/HOL [95].

Isabelle can be used under Linux or Windows. The development interface, which is the most recommended, is based on the Xemacs text editor. Figure 8.1 presents the interface of the environment in which the embedding of our model is performed.

8.4 Embedding the Semantic Model

In what follows, we present an overview of the embedding of our semantic model. More precisely, we consider in this embedding two kinds of actions: `Lock` and `Unlock`. These actions are performed on objects, which are considered as resources. The resource mapping between actions and resources is represented by a constant called **`resMap`**. A dependence map is embedded as a list of infinite branching graphs that captures the dependence between events, which are embedded as occurrences of actions. An infinite branching graph is represented by a datatype called `M`. The prefix relation between


Isabelle/Isar Proof General: BytecodeNoBack.thy
File Edit View Cmds Tools Options Buffers Proof-General Isabelle/Isar Help
BytecodeNoBack.thy | Traces.thy | *isabelle/isar*

```
done

lemma listsize [simp]:
"length (snd (exec ins d bl p)) \<le> length bl"
apply(induct_tac ins)
apply(simp_all)
apply(induct_tac[!] d)
apply(auto)
done
Raw-----XEmacs: BytecodeNoBack.thy      (Isar script Font Scripting )----54%----------------------------
proof (prove): step 1

fixed variables: ins, d, bl, p

goal (lemma (listsize), 17 subgoals):
 1. !!nat::nat. Nat.size (snd (BytecodeNoBack.exec (BytecodeNoBack.bytecode.ILOAD nat) d bl p)) <= Nat.si
ze bl
 2. !!nat::nat. Nat.size (snd (BytecodeNoBack.exec (BytecodeNoBack.bytecode.ISTORE nat) d bl p)) <= Nat.s
ize bl
 3. !!int::IntDef.int. Nat.size (snd (BytecodeNoBack.exec (BytecodeNoBack.bytecode.ICONST int) d bl p)) <
= Nat.size bl
 4. !!int::IntDef.int. Nat.size (snd (BytecodeNoBack.exec (BytecodeNoBack.bytecode.BIPUSH int) d bl p)) <
= Nat.size bl
```

Figure 8.1: Development Environment

two dependence maps is established using a function called F, which is defined co-inductively, while the deterministic process space is represented by a set called `C`.

The embedding of the main features of our model is as follows:

```
typedecl classType
datatype fields = "int list"
types    lockCounter = "int"
types    object = "classType × fields × lockCounter"
types    resources = "object set"
datatype action =  Lock object |
                   Unlock object
types    event = "action × nat"
consts   resMap :: "action ⇒ resources"
types    PEvent = "event set × event"
datatype M = Bot | Node PEvent "nat ⇒ M"
types    lab = "(event × action) set"
types    process = "(M list × lab) × resources"

consts   res :: "process ⇒ resources"
recdef   res "{}"
"res P = (⋃ e ∈ Domain (snd (fst P)). resMap (fst e)) ∪ (snd P)"

consts   resdiff :: "process × process ⇒ resources"
recdef   resdiff "{}"
"resdiff (P,Q) =
(⋃ e ∈ (Domain (snd (fst P)) - Domain (snd (fst Q))). resMap (fst e))"
```

```
consts dom :: "M ⇒ 'a set"
defs dom_def:
"dom M == (case M of Bot ⇒ {} | (Node a F) ⇒ {x. ∃ y. y = F x})"

(*Prefix relation*)
consts F :: "(M × M) set ⇒ (M × M) set"
coinductive "F(Q)"
  intros
    BOT_I:
      "(Bot,T) ∈ F(Q)"
    TRACE_I:
      "[|a = b; M = Node a G ; N = Node b H; ∀ i ∈ dom(M).
         ∃ j ∈ dom(N). (G i,H j) ∈ Q |] ⇒ (M,N) ∈ F(Q)"

consts prefixRel :: "M list × M list ⇒ bool"
defs   prefixRel_def:
"prefixRel R S == ∀ x ∈ set R. ∃ y ∈ set S. ∃ H. (x,y) ∈ F(H)"

(*First healthiness condition*)
consts  por :: "'a list ⇒ bool"
primrec "por [] = True"
        "por (l#ls) = (if (antisym l) then (por ls) else False)"

constdefs C :: "(process) set"
"C == {x. por (fst (fst x))}"
```

After the embedding of the main features of our semantic model, we provide the subset of the JVML/CLDC language for which we provide later a denotational semantics based on this model.

8.5 JVML/CLDC Subset Syntax

Hereafter, we present the JVML/CLDC subset, which we consider in this embedding.

```
  bytecode   =  ILOAD     index
             |  ISTORE    index
             |  ICONST    val
             |  BIPUSH    val
             |  IADD
             |  IMUL
             |  IFEQ      index
             |  IFICMPEQ  index
             |  IFICMPNE  index
             |  IFICMPLE  index
             | GOTOF      index
             | RETURN
             | IRETURN
             | ALOAD      index
             | ASTORE     index
             | DUP
             | MONITORENTER
             | MONITOREXIT
```

The informal semantics of each bytecode is as follows:

- ILOAD index: the value of the local variable at the position "index" of the local variable table is pushed into the stack. The ALOAD bytecode has the same semantics but the loaded value is an address of an object.
- ISTORE index: the value on top of the stack is stored in the local variable at the position "index" of the local variable table, then popped from the stack. The ASTORE bytecode has the same semantics but the stored value is an address of an object.
- ICONST val: the constant value "val" is pushed into the stack.
- BIPUSH val: the value "val" is pushed into the stack after its conversion to an integer.
- DUP: the value that figures on the stack is duplicated.
- IADD: the two values that figure on top of the stack are popped in temporary variables then added. The result is pushed into the stack.
- IMUL: the two values that figure on top of the stack are popped in temporary variables then multiplied. The result of the multiplication is pushed into the stack.
- IFEQ index: a conditional branch to the instruction at the position "index" of the program. The jump is performed if the top of the stack is equal to 0.
- IFICMPEQ index: a conditional branch to the instruction at the position "index" of the program. The jump is performed if the two values on top of the stack are equal.
- IFICMPNE index: a conditional branch to the instruction at the position "index" of the program. The jump is performed if the two values on top of the stack are not equal.

- IFICMPLE index: a conditional branch to the instruction at the position "index" of the program. The jump is performed if the value on top of the stack is equal or greater than the value that is just below it on the stack.
- GOTOF index: an unconditional branch to the instruction at the position "index" of the program.
- RETURN: there is no returned value by the executed method.
- IRETURN: the integer value, which is on top of the stack is returned.
- MONITORENTER: the current thread tries to lock the object on top of the stack. If this is possible, the lock counter is incremented by one. Otherwise this thread waits the release of the lock.
- MONITOREXIT: the current thread tries to unlock the object. If this is possible, the lock counter is decremented by one. If the lock counter reaches zero, the object is released.

8.6 A Case Study about the Validation of Optimizations of JVML/CLDC Programs

In this section, we present some basic optimizations that can be performed on JVML/-CLDC programs. To clarify the presentation, we present first the Java source program, which corresponds to the JVML/CLDC program on which the optimization can be performed. We describe how the optimization is done on the source program and we suppose there exists a similar transformation that can be performed on its compilation output i.e. this optimization can be performed on the compiled file, which is a JVML/CLDC program. Note that the representation of JVML/CLDC programs in Isabelle/HOL is the result of an abstraction at the instruction level. Moreover, we deal here just with intraprocedural optimizations i.e. optimizations that are performed inside one method.

8.6.1 Constant Propagation

A constant propagation transformation is performed if a variable is assigned to a constant value. If there is no other future assignment of this variable, it can be replaced by the constant value in order to avoid more computations. The following Java method contains a code on which compilers can perform constant propagation.

```
public int foo() {
    int x,y;
    x = 3;
    y = x + 4;
    return y;
}
```

The definition in Isabelle/HOL of the JVML/CLDC code, which is the compilation output of the already presented Java code, is the following:

```
constdefs orprog1 :: "prog"
"orprog1  == [ICONST 3, ISTORE 1, ILOAD 1, ICONST 4, IADD, ISTORE 2, ILOAD 2,
              IRETURN]"
```

In the Java source, we see that the variable x can be replaced by 3 in the expression x + 4. The optimized Java code is the following:

```
public int foo() {
    int x,y;
    x = 3;
    y = 7;
    return y;
}
```

The associated compiled code is the following:

```
constdefs opprog1 :: "prog"
"opprog1 == [ICONST 3, ISTORE 1, BIPUSH 7, ISTORE 2, ILOAD 2, IRETURN]"
```

As mentioned previously, we assume that we have a transformation between orprog1 and opprog1, which is a constant propagation optimization. For instance, detecting that the variable x is constant at the bytecode level can be done as follows: there is no store instruction after the instruction ISTORE 1. This means that the value of the variable at position 1 (which is "x") is not changed. So any load of this variable, coming after this store instruction, can be replaced by the value of "x" i.e. the instruction ILOAD 1 is replaced by ICONST 3[4]. In a second step, the addition is performed and the sequence "ICONST 3,ICONST 4,IADD" is replaced by BIPUSH 7. This scenario is an example that shows how the constant propagation can be performed at the bytecode level. More fancy analysis can also be used to perform this optimization.

Proving that this constant propagation optimization is correct means that orprog1 and opprog1 are associated with the same denotation. This is proved later when we present the semantics.

[4]For a subset without backward jumps

8.6.2 Dead Assignment Elimination

A dead assignment elimination is an optimization targeting the removal of dead variables. These variables are never used after their assignment.

The following Java code contains a dead assignment for the variable "x" in the statement "x = 3".

```
public int foo() {
    int x,y;
    y = 0;
    x = 3;
    y = y + 2;
    return y;
}
```

The associated compiled code, as presented in Isabelle, is the following:

```
constdefs orprog2 :: "prog"
"orprog2  == [ICONST 0, ISTORE 2, ICONST 3, ISTORE 1, ILOAD 2, ICONST 2, IADD,
              ISTORE 2, ILOAD 2, IRETURN]"
```

The removal of this statement, gives the following code:

```
public int foo() {
    int x,y;
    y = 0;
    y = y + 2;
    return y;
}
```

The associated compiled code is the following:

```
constdefs opprog2 :: "prog"
"opprog2  == [ICONST 0, ISTORE 2, ILOAD 2, ICONST 2, IADD, ISTORE 2, ILOAD 2,
              IRETURN]"
```

8.6.3 Common Subexpression Elimination

Common subexpression elimination consists of finding a redundant expression and replacing this expression with a temporary variable in which the evaluation of the expression is stored. Future uses of this expression are replaced by the temporary variable.

The following Java code shows a redundant expression "a + b". This expression is assigned to the variable "x" and used again in the expression "y > a + b".

```
public int foo() {
    int a,b;
    int x,y;
    a = -2;
    b = 3;
    x = a + b;
    y = a*b;
    if (y > a+b) return 0;
    else  return 1;
}
```

The representation of the associated compiled code is the following:

```
constdefs orprog3 :: "prog"
"orprog3 == [BIPUSH -2, ISTORE 1, ICONST 3, ISTORE 2, ILOAD 1, ILOAD 2, IADD,
             ISTORE 3, ILOAD 1, ILOAD 2, IMUL, ISTORE 4, ILOAD 4, ILOAD 1, ILOAD 2,
             IADD, IFICMPLE 19, ICONST 0, IRETURN, ICONST 1, IRETURN]"
```

To avoid redundant computation of "a + b", optimizing compilers can replace the expression "y > a + b" with "y > x"[5]. The following Java code is the output of this optimization:

```
public int foo() {
    int a,b;
    int x,y;
    a = -2;
    b = 3;
    x = a + b;
    y = a*b;
    if (y > x) return 0;
    else  return 1;
}
```

The representation of the associated compiled code in Isabelle is the following:

```
constdefs opprog3 :: "prog"
"opprog3 == [BIPUSH -2, ISTORE 1, ICONST 3, ISTORE 2, ILOAD 1, ILOAD 2, IADD,
             ISTORE 3, ILOAD 1, ILOAD 2, IMUL, ISTORE 4, ILOAD 4, ILOAD 3,
             IFICMPLE 17, ICONST 0, IRETURN, ICONST 1, IRETURN]"
```

[5]This is better than using a temporary variable

8.6.4 Denotational Semantics

In the sequel, we present the embedding of a denotational semantics for a subset of JVML/CLDC. The studied subset excludes backward jumps. Note that an entire research thesis targeted studying just loops in JVML programs [108] using Isabelle. Moreover, we suppose that we are in the context of just one thread to avoid more technical problems. First, we give a version excluding the use of our semantic domain in order to be able to execute the specification. Then, we provide the extension of this semantics with our process space definition.

Semantics of one Bytecode

The semantics of one bytecode is specified by a semantic function called `exec`. This function computes a pair composed of a denotation and a continuation for one bytecode. The *denotation* is a tuple composed of a stack, a local variable table and the returned result. The continuation represents the remaining code of the program to be executed. Before presenting the embedding of the semantic function `exec`, we present the embedding of the language syntax.

```
types index = nat types
val = int
types stack = "val list"
types locvars = "val list"

bytecode    =  ILOAD     index
            |  ISTORE    index
            |  ICONST    val
            |  BIPUSH    val
            |  IADD
            |  IMUL
            |  IFEQ      index
            |  IFICMPEQ  index
            |  IFICMPNE  index
            |  IFICMPLE  index
            |  GOTOF     index
            |  ALOAD     index
            |  ASTORE    index
            |  IRETURN

(*A Program is a list of bytecodes*)
types prog = "bytecode list"
```

In what follows, we present the embedding of the semantic function `exec` in Isabelle/HOL.

```
datatype result = NoValue | Value "int"

types denotation = "stack × locvars × result"

(*Execution continuation*)
types cont = "prog"

consts exec :: "bytecode ⇒ denotation ⇒ cont ⇒ prog ⇒ denotation × cont"
primrec
"exec (ILOAD  ind)  d  c p =
(let (sk,lv,rs) = d
 in  (((lv ! ind) # sk,lv,rs)),c)"

"exec (ISTORE ind)  d  c p =
(let (sk,lv,rs) = d
 in  (((tl sk),lv[ind:=hd sk],rs)),c)"

"exec (ICONST v)    d  c p =
(let (sk,lv,rs) = d
 in  ((v#sk,lv,rs)),c)"

"exec (BIPUSH v)    d  c p =
(let (sk,lv,rs) = d
 in  ((v#sk,lv,rs)),c)"

"exec IADD          d  c p =
(let (sk,lv,rs) = d;
      (a,b)=(hd sk, hd(tl sk));
       v = a + b
 in  ((v#(tl(tl sk)),lv,rs),c))"

"exec IMUL          d  c p =
(let (sk,lv,rs) = d;
      (a,b)=(hd sk, hd(tl sk));
       v = a * b
 in  ((v#(tl(tl sk)),lv,rs),c))"

"exec (IFEQ ind)   d  c p =
(let (sk,lv,rs) = d;
      v = hd sk;
      v1 = (length p) - (length c);
      disp = (ind - v1)
 in   if (v ≠ 0)
      then ((tl sk,lv,rs),c)
      else ((tl sk,lv,rs),drop disp c))"
```

```
"exec (IFICMPEQ ind) d c p =
(let (sk,lv,rs) = d;
     v1 = hd sk;
     v2 = hd (tl sk)
 in  if (v1 ≠ v2)
     then  ((tl (tl sk),lv,rs),c)
     else  (let v = (length p) - (length c);
                disp = (ind - v)
            in  ((tl (tl sk),lv,rs),drop disp c)))"

"exec (IFICMPNE ind) d c p =
(let (sk,lv,rs) = d;
      v1 = hd sk;
      v2 = hd (tl sk)
 in  if (v1 ≠ v2)
     then  (let v = (length p) - (length c);
                disp = (ind - v)
            in  ((tl (tl sk),lv,rs),drop disp c))
     else  ((tl (tl sk),lv,rs),c))"

"exec (IFICMPLE ind) d c p =
(let (sk,lv,rs) = d;
      v1 = hd sk;
      v2 = hd (tl sk)
 in  if (v2 ≤ v1)
     then (let v = (length p) - (length c);
               disp = (ind - v)
           in  ((tl (tl sk),lv,rs),drop disp c))
     else ((tl (tl sk),lv,rs),c))"

"exec (GOTOF ind)   d  c p =
(let  v = (length p) - (length c);
      disp = (ind - v)
 in  (d,drop disp c))"

"exec (ALOAD  ind)  d  c p =
(let (sk,lv,rs) = d
 in  (((lv ! ind) # sk,lv,rs)),c)"

"exec (ASTORE ind)  d  c p =
(let (sk,lv,rs) = d
 in  (((tl sk),lv[ind:=hd sk],rs)),c)"

"exec RETURN       d  c p =
(let (sk,lv,rs) = d
 in ((sk,lv,NoValue),c))"
```

```
"exec IRETURN        d  c p =
(let (sk,lv,rs) = d ;
      v = hd sk
 in  ((tl sk,lv,Value v),c))"
```

JVML/CLDC Program Semantics

Before providing the semantics of a JVML/CLDC program, we present two lemmas that prove that the continuation argument is decreasing in size during execution. In these lemmas, induction, auto tactics and simplification are needed.

```
declare Let_def[simp] option.split[split]

lemma listsize_eq: "(length (snd (exec ins d bl p)) < Suc (length bl)) =
                    (length (snd (exec ins d bl p)) ≤ length bl)"
apply(auto)
done

lemma listsize [simp]: "length (snd (exec ins d bl p)) ≤ length bl"
apply(induct_tac ins)
apply(simp_all)
apply(induct_tac[!] d)
apply(auto)
done
```

The semantics of a JVML/CLDC program is specified by a semantic interpretation function called sem, which is defined recursively and which uses the function exec. Its termination depends on the size of the first argument that refers to the continuation and which should decrease. To help the prover doing the full proof, we give a "hint" that suggests the use of the previous lemmas as simplification rules. In the sequel, we present the embedding of the function sem.

```
consts  sem :: "prog × denotation × prog ⇒ denotation"
recdef sem "measure (λ(xs,a). length xs)"

"sem ([],d,p)  = d"

"sem ((b#bl),d,p) = (sem (snd (exec b d bl p), fst (exec b d bl p),p))"

(hints recdef_simp: listsize_eq)
```

Equivalence Relations

In the following, we provide two possible relations that can be used to establish the semantic equivalence between JVML/CLDC programs:

- *Strong* equivalence: the original and optimized program are strongly equivalent if they are associated with the same denotation i.e. are associated to the same stack, the same local variable table and return the same value after their execution. As defined, this relation is too restrictive. In fact, two equivalent programs can return the same value without doing the same treatments on the stack or without having the same local variable table. Henceforth, we provide another definition of an equivalence relation, which is a weakened form of the strong equivalence.
- *Weak* equivalence: the original and optimized programs are weakly equivalent if they return the same value when they are executed.

Hereafter, we suppose that the stack, the local variable table are empty before the execution of any program:

```
constdefs d:: "denotation"
"d ==([0,0,0,0,0,0,0],[0,0,0,0,0,0,0],NoValue)"
```

The strong equivalence is defined in Isabelle/HOL as follows:

```
constdefs equivalent :: "prog ⇒ prog ⇒ bool"

"equivalent p1 p2 == (sem (p1,d,p1) = sem (p2,d,p2))"
```

The weak equivalence is defined in Isabelle/HOL as follows:

```
constdefs weakequivalent :: "prog ⇒ prog ⇒ bool"

"weakequivalent p1 p2 ==
(snd (snd (sem (p1,d,p1))) = snd (snd (sem (p2,d,p2))))"
```

Executing the Specification by Code Generation

Isabelle offers a way to execute a specification. This means, in our case, that given a JVML/CLDC program, we can see the output of this program. Note that the code generation capabilities of Isabelle (version 2004) are limited. More development of this feature will be incorporated in future versions.

The code generation of the semantics of orprog1 and opprog1 is specified in Isabelle as follows:

```
generate_code
  test1 = "sem (orprog1,d,orprog1)"
  test2 = "sem (opprog1,d,opprog1)"
  test3 = "equivalent orprog1 opprog1"
```

The execution of these tests under ML is defined as follows:

```
ML * test1 *
ML * test2 *
ML * test3  *
```

The output of test1 is the following:

```
val it =
([0,0,0,0,0,0,0],([0,3,7,0,0,0,0],Value 7))
 : int list * (int list * result)
```

The output of test2 is the following:

```
val it =
([0,0,0,0,0,0,0],([0,3,7,0,0,0,0],Value 7))
 : int list * (int list * result)
```

The strong equivalence testing of the original and the optimized program provides the following result:

```
val it = true : bool
```

Now, we consider the following Java code:

```
public int foo() {
    int x,y;
    int b = 6;
    if (b != 5)
    {
    x = 3;
    y = x + 4;
    }
    else
    y = 2;
    return y;
}
```

The representation of the associated compiled code for this program in Isabelle is the following:

```
constdefs branchprog1 :: "prog"
"branchprog1 == [BIPUSH 6, ISTORE 3, ILOAD 3, ICONST 5, IFICMPEQ 12, ICONST 3,
                 ISTORE 1, ILOAD 1, ICONST 4, IADD, ISTORE 2, GOTOF 14, ICONST 2,
                 ISTORE 2, ILOAD 2, IRETURN]"
```

The semantics of this program is the following:

```
val it =
([0,0,0,0,0,0,0],([0,3,7,6,0,0,0],Value 7))
  : int list * (int list * result)
```

This program returns the same value as the program orprog1. However, the strong equivalence test returns false i.e. it considers these programs to be not semantically equivalent. The strong equivalence test rejects these two programs since the two programs modify in different ways the local variable table. However, the weak equivalence test succeeds.

The programs orprog2 and opprog2 are weakly equivalent. In fact, the semantics of orprog2 is the following:

```
val it =
([0,0,0,0,0,0,0],([0,3,2,0,0,0,0],Value 2))
  : int list * (int list * result)
```

While the semantics of oprpog2 is the following:

```
val it =
([0,0,0,0,0,0,0],([0,0,2,0,0,0,0],Value 2))
  : int list * (int list * result)
```

The programs orprog3 and opprog3 have the same semantics:

```
val it =
([0,0,0,0,0,0,0],([0,~2,3,1,~6,0,0],Value 1))
  : int list * (int list * result)
```

The programs orprog3 and opprog3 are strongly equivalent.

The presented examples show that the equivalence relation depends indeed on the performed optimization.

Extending the Semantics with Synchronization Bytecodes

In this section, we modify the semantics by handling two important bytecodes, which are: `MONITORENTER` and `MONITOREXIT`, which are meant to perform synchronization operations. The strong equivalence relation now requires also that the original and optimized programs do the same lock/unlock actions on the same objects and the two heaps are the same after the execution of these programs. Here, we consider that we observe just actions that lock or unlock objects but it is worth to mention that other abstractions can be considered. In fact, we can observe exceptions or communications. This means that two programs are strongly equivalent if they lock and unlock the same objects or/and throw the same exceptions or/and do the same communications and are associated with the same stack and local variable table and return the same result after their execution.

The grammar of actions is already presented in Section 8.4. The denotation is extended with two types: `process` and `heap`. The latter is specified as a partial map between addresses and objects.

Hereafter, we present the datatypes that are needed to elaborate the extended version of the semantics.

```
types stack = "val list"

types locvars = "val list"

types heap ="int ⇒ object option"

types prog = "bytecode list"

datatype result = NoValue | Value "int"

types denotation = "process × heap × stack × locvars × result"

types cont = "prog"
```

The embedding of the semantic function exec is as follows.

```
consts exec :: "bytecode ⇒ denotation ⇒ cont ⇒ prog ⇒ denotation × cont"
primrec

"exec (ILOAD  ind)  d  c p =
(let (pr,he,sk,lv,rs) = d
 in  ((seqcomp (pr,(([],{}),{})),he,(lv ! ind) # sk,lv,rs),c))"

"exec (IFEQ ind)   d  c p =
(let (pr,he,sk,lv,rs) = d;
      v = hd sk;
      v1 = (length p) - (length c);
      disp = (ind - v1)
 in   if (v ≠ 0)
      then ((seqcomp (pr,(([],{}),{})),he,tl sk,lv,rs),c)
      else ((seqcomp (pr,(([],{}),{})),he,tl sk,lv,rs),drop disp c))"

"exec MONITORENTER  d  c p =
(let (pr,he,sk,lv,rs) = d ;
     v = hd sk
 in  (case he(v) of
           None ⇒ (d,[]) |
           Some (cl,fs,lc) ⇒
                (let lc1 = lc + 1;
                     he1 = he(v |-> (cl,fs,lc1));
                     pr1 =(([Node ({},(Lock (he v),1)) (λ i. Bot)],
                            {((Lock (he v),1),Lock (he v))}),{})
                 in  ((seqcomp (pr,pr1),he1, tl sk,lv,rs),c))))"
```

```
"exec MONITOREXIT   d  c p =
(let (pr,he,sk,lv,rs) = d ;
      v = hd sk
 in  (case he(v) of
           None ⇒(d,[]) |
           Some (cl,fs,lc) ⇒
                (let lc1 = lc - 1;
                     he1 = he(v |-> (cl,fs,lc1));
                     pr1 =  (([Node ({},(Unlock (he v),1)) (λ i. Bot)],
                             {((Unlock (he v),1),Unlock (he v))}),{})
                 in  ((seqcomp (pr,pr1),he1,tl sk,lv,rs),c))))"
```

The function seqcomp performs a sequential composition of two processes. This is the embedding of the strict sequential composition semantic operator, which is defined in the previous chapter. It is embedded as follows:

```
consts    seqNodes :: "M ⇒ M ⇒ event set ⇒ M"
primrec
"seqNodes Bot H S = H"
"seqNodes  (Node a G) H S =
(if (G = (λ i. Bot))
 then (case H of Bot ⇒ (Node a G) |
                Node (P,e) K ⇒ (Node a (fun_upd G 0 (Node (P ∪ S,e) K)))
 else (Node a (λ i. (case H of Bot ⇒ (G i) |
                               Node (P,e) K ⇒
                                      (case (G i) of
                                            Bot ⇒ Bot |
                                            (Node b T) ⇒
                                                 (let (T = Node (P ∪ S,e) K)
                                                  in (seqNodes (G i) T S)))))))"

consts    seqMap :: "M ⇒ M list ⇒ event set ⇒ M list"
primrec
"seqMap x [] S = [x]"

"seqMap x (y#ys) S = (seqNodes x y S)#(seqMap x ys S)"

consts final  :: "M ⇒ event set"
primrec
"final Bot = {}"
"final (Node a G) =  (if (G = (λ i. Bot)) then snd a
                                          else (⋃ i. final (G i)))"
```

```
consts terminals :: "M list ⇒ event set"
primrec
"terminals [] = {}"
"terminals (x#xs) = (final x Un terminals xs)"

consts    seqMapList :: "M list  ⇒ M list ⇒ event set ⇒ M list"
primrec
"seqMapList [] T S  = T"
"seqMapList (x#xs) T S  = concat [(seqMap x T S),(seqMapList xs T S)]"

consts    seqcomp :: "process × process ⇒ process"
recdef    seqcomp "{}"
"seqcomp (P,Q) =
(if (snd P = {})
 then  ((seqMapList (fst (fst P)) (fst (fst Q))(terminals (fst (fst P))),
       (snd (fst P)) ∪ (snd (fst Q))),snd Q)
 else  (fst P, (snd P) ∪ (res Q)))"
```

Note that a terminal node has no successor. In our embedding, the function λ i. Bot is used to mark a terminal node. Note also that the semantics of MONITORENTER and MONITOREXIT are more complicated than what presented. In fact, the exception NullPointerExpection is raised if the synchronized object, which figures on top of the stack, is null. Moreover, in the case of multithreaded programs, waiting and notification mechanisms are used to manage the synchronization process. In our semantics, if the synchronized object is null, the program terminates, at this verification point, by having an empty list of instructions as continuation. Moreover, we consider monothreaded programs, avoiding by this more technical issues related to the embedding of waiting and notification mechanisms. The semantics of MONITORENTER and MONITOREXIT are provided just to show the emergence of observable actions in the denotation and the use of our semantic model to prove program equivalence.

In the sequel, we present two programs that are strongly equivalent. In fact, they are associated with the same dependence map, the same heap, the same stack, local variable table and return the same result.

```
constdefs orprog4 :: "prog"

"orprog4  == [ALOAD 0, DUP, ASTORE 3, MONITORENTER, ICONST 3, ISTORE 1, ILOAD 1,
              ICONST 4, IADD, ISTORE 2, ILOAD 2, ALOAD 3, MONITOREXIT, IRETURN]"
```

```
constdefs opprog4 :: "prog"

"opprog4 == [ALOAD 0, DUP, ASTORE 3, MONITORENTER, ICONST 3, ISTORE 1, BIPUSH 7,
             ISTORE 2, ILOAD  2, ALOAD 3, MONITOREXIT, IRETURN]"
```

The program opprog4 is the transformation of orprog4 by constant propagation. These two programs are associated with the same following denotation:

```
val it =
((([Node ({},(Lock this,1)) (fun_upd (λ i. Bot) 0 Node ({(Lock this,1)},
   (Unlock this,1)) (λ i. Bot))],{((Lock this,1),Lock this),
   ((Unlock this,1),Unlock this)}),{}), map_of [(0,this)],
   [0,0,0,0,0,0,0],([0,3,7,6,0,0,0],Value 7))
  : int list * (int list * result)
```

Note that the variable "this" refers to the instance on which a given method was called. We suppose that this object is stored at the memory address zero to simplify the presentation.

8.7 Conclusion

In this chapter, we provided a case study that shows how our model can be embedded in a theorem prover such as Isabelle. Moreover, we defined some definitions of equivalence relations between programs, which are coded in a subset of JVML/CLDC. We discussed the impact of their definitions on the correctness of optimizations. We think that the equivalence relation should be defined with respect to the specificities of the optimization. These specificities allow to know the required abstractions to be performed in order to prove that original and optimized programs have the same semantics.

Chapter 9

Conclusion

In the current decade, we witness a wide proliferation of embedded devices. In this context, Java is the defacto standard language for developing mobile and embedded applications. Thanks to the platform J2ME/CLDC, Sun Microsystems is becoming a major player in the world of embedded and mobile computing. An important fact that corroborates this statement is that more than 1 billion of Java-enabled phones will be deployed in the market [84]. A successful deployment of Java, on these devices, relies on a fast and lightweight execution environment. Our research provides practical and theoretical solutions for the design, implementation and validation of the acceleration techniques of the defacto standard embedded Java platform: J2ME/CLDC.

At the practical level, we provided a design and an implementation of a fast and lightweight dynamic compiler for J2ME/CLDC. The compiler is built on top of KVM, which is the virtual machine coming with J2ME/CLDC. Our compiler is the first academic work that targets CLDC Java virtual machines optimization by dynamic compilation. The other unpublished compilation techniques are integrated in commercial products such as CLDC HotSpot [84] and Jbed [113]. Moreover, our solution, besides the compilation of all kinds of bytecodes, covers several issues related to the integration of a dynamic compiler into a virtual machine such as multi-threading support, exception handling, garbage collection, switching mechanism between the compiler and the interpreter modes, etc. Furthermore, the results show that the optimized virtual machine is fast. In fact, our solution allows to improve the performance of KVM by a factor of 4 while the memory footprint does not exceed 138 KB. In another practical aspect of this research, we presented a design of a unified strategy that combines fast interpretation and lightweight compilation together with a fast, dynamic and flexible acceleration technique for the method lookup mechanism. The aforementioned designed techniques together with the E-Bunny compiler constitute our practical contribution.

At the theoretical side, we provided a semantic model for true concurrency with unbounded non-determinism. This semantic model is currently accommodated to JVML/-CLDC in the thesis of Ms Lamia Ketari [66] and is intended to be used to validate optimizations of JVML/CLDC programs. The model is denotational and rests on an extension of the resource pomsets semantics of Gastin and Mislove [50]. We presented the construction of the process space and exhibited its algebraic properties. Moreover, we provided the semantic interpretation of some useful concurrency operators. As a downstream result, this led to a fixpoint semantics of recursion.

We also provided an interesting overview about the validation of JVML/CLDC program optimizations using the theorem prover Isabelle. The use of a theorem proving tool allows to automate, to some extent, the validation of optimizations. Moreover, by having machine-checked proofs, we avoid possible faulty proofs. The motivation behind using Isabelle/HOL is that it was widely used for providing a semantics for Java or JVML/CLDC. In fact, many prominent researchers such as Tobias Nipkow [94], David Von Oheimb [97] used Isabelle/HOL for providing semantics for small subsets of Java and JVML/CLDC. In these projects, the elaborated semantics are operational. Our strategy is based on a denotational approach. More precisely, we presented an embedding of our deterministic process space and a sketch of a denotational semantics for a subset of JVML/CLDC. The objective of this embedding is to show that our model can be used for optimization validation.

Future Works There are some possible improvements that can be introduced as complementary solutions to what we proposed. More precisely, the following enhancements can be considered:

- The design and implementation of static analysis techniques (offline) in order to enhance the prediction of hotspot methods. For instance, the static detection of loops can improve the compilation rate of hotspot methods. These techniques can be relevant for J2ME APIs since the source code of these applications is available.
- The design and implementation of fast and lightweight multi-level optimizing compilers in order to enhance the generated code quality and tune the degree of optimizations. The memory and power limitations of embedded devices should be taken into consideration.
- The design and implementation of compact data structures for storing information about JVM programs such as the structure of objects, threads and the constant pool. This will minimize the required memory.

Moreover, it is worth to investigate the following research directions, which will pave the way for a fully-fledged acceleration validation framework for JVML/CLDC:

- The full embedding of our semantic model for JVML/CLDC in Isabelle, which includes recursion-related issues. This requires the embedding of the theory of local cpos in Isabelle.
- The investigation of more case studies about the validation of JVML/CLDC program optimizations.

We think that providing a fully-fledged semi-automatic environment for the validation of optimizations of object-oriented programs and specifically of JVML/CLDC programs is a very interesting future challenge. In fact, many aspects such as exception handling, method lookup and multi-threading are, without any doubt, important issues to be investigated. We think that these issues are very interesting for the research community since there is no work, in our best knowledge, that targets the use of theorem provers in order to validate optimizations of JVML/CLDC programs. Our embedding is a step towards handling these issues.

Finally, our future work will be oriented mainly to provide a complete framework in which we can validate automatically or systematically several kinds of JVML/CLDC program optimizations and also prove the correctness of some mechanisms, which are implemented in a Java virtual machine. For instance, the embedding of our semantic model in Isabelle can be used to verify the correctness of the exception handling mechanism.

Bibliography

[1] Intel VTune Performance Analyzer 7.1. `http://www.intel.com`.

[2] S. Abramsky and A. Jung. Domain Theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[4] B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, S. Hummel M. Hind, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[5] J. Alves-Foss and F. Lam. Dynamic denotational semantics of java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 201–240. Springer-Verlag, June 1999.

[6] K. Apt and G. Plotkin. Countable Non-Determinism and Random Assignment. *Journal of the ACM (JACM)*, 33(4):724–767, January 1986.

[7] ARM. `http://www.arm.com`.

[8] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. Adaptive Optimization in the Jalapeño JVM. *ACM SIGPLAN Notices*, 35(10):47–65, October 2000.

[9] M. Arnold, M. Hind, and B. Ryder. An Empirical Study of Selective Optimization. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, volume 2017, pages 49–67, Yorktown Heights, New York, USA, August 2000. Lecture Notes in Computer Science.

[10] A. Azevedo, A. Nicoleau, and J. Hummel. Java Annotation-aware Just-in-Time (AJIT) Compilation System. In *Proceedings of the ACM Java Grande Conference*

(JAVA'99), pages 142–151, San Francisco, California, USA, June 1999. ACM Press.

[11] M. Azzam. A selective dynamic compiler for embedded java virtual machines targeting arm processors. Master's thesis, Computer Sience and Software Enginnering Department, Laval University, 2004.

[12] D. Bacon, S. Fink, and D. Grove. Space and Time-Efficient Implementation of the Java Object Model. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP'02)*, volume 2374 of *Lecture Notes in Computer Science*, pages 111–132, Malaga, Spain, June 2002. Springer-Verlag.

[13] M. Bednarczyk. *Categories of Asynchronous Systems.* PhD thesis, University of Sussex, 1988.

[14] J. Blech, S. Glesner, J. Leitner, and S. Mlling. Optimizing Code Generation from SSA Form: A Comparison Between Two Formal Correctness Proofs in Isabelle/HOL. In *Proceedings of the Workshop on Compiler Optimization meets Compiler Verification (COCV'05)*, Edinburgh, Scotland, April 2005. Elsevier.

[15] D. Bolignano and M. Debbabi. A Semantic Theory for Concurrent ML. In *Proceedings of the International Conference on Theoretical Aspects of Computer Software (TACS'94)*, volume 789 of *Lecture Notes in Computer Science*, pages 766–785, Sendai, Japan, April 1994. Springer-Verlag.

[16] H. Boom. A Weaker Precondition for Loops. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4:668–677, October 1982.

[17] E. Borger and W. Schulte. A Programmer Friendly Modular Definition of the Semantics of Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 353–404. Springer-Verlag, June 1999.

[18] S. Brookes, C. Hoare, and A. Roscoe. An Improved Failures Model for Communicating Processes. In *Seminar on Concurrency, Carnegie-Mellon University*, volume 197, pages 281–305. Springer-Verlag, 1985.

[19] S. Brooks, C. Hoare, and A. Roscoe. A Theory for Communicating Sequential Processes. *JACM*, 31(3):560–599, January 1984.

[20] D. Buytaert, F. Arickx, and J. Acunia. A Profiler and Compiler for the Wonka Virtual Machines. In *Works-in-Progress Session of the 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, San Francisco, CA, USA, August 2002. USENIX Association.

[21] Bytecodes. Method Call Overhead. `http://www.bytecodes.com`, 2003.

[22] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An Event-Based Structural Oprational Semantics of Multi-Threaded Java. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 157–200. Springer-Verlag, June 1999.

[23] G. Chaitin, M. Auslander, A. Chandra, J. Cocke, M. Hopkins, and P. Markstein. Register Allocation Via Coloring. *Computer Languages*, 6:47–57, January 1981.

[24] Chicory. `www.chicorysystems.com`.

[25] M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Languages Design and Implementation (PLDI'00)*, pages 13–26, Vancouver, Canada, June 2000. ACM Press.

[26] G. Comeau. Java Companion Processors versus Accelerators. `http://www.e-sim.com/corporate/press/year_2000/000501.htm`.

[27] Nazomi Communications. Bootsing the Performance of Java Software on Smart Handheld Devices and Internet Appliance. `http://www.nazomi.com`.

[28] Imagix Corporation. `http://www.imagix.com`.

[29] Intel Corporation. Intel Architecture Software Developer's Manual, September 1997. White paper.

[30] Standard Performance Evaluation Corporation. SPEC JVM98 Benchmarks. `http://www.specbench.org/osg/jvm98/`, 1998.

[31] B. Cox. *Object-Oriented Programming, An Evolutionary Approach.* Addison-Wesley, MA, USA, 1987.

[32] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17(3):36–43, May 1997.

[33] D. Bacon and R. Konuru and C. Murthy and M. Serrano. Thin Locks: Featherweight Synchronization for Java. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 258–268, Montreal, Canada, June 1998.

[34] J. Dean, D. Grove, and C. Chambers. Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Science*, pages 77–101, Arhus, Denmark, August 1995. Springer-Verlag.

[35] M. Debbabi, M. Erhioui, L. Ketari, N. Tawbi, H. Yahyaoui, and S. Zhioua. Method Call Acceleration in Embedded Java Virtual Machines. In *Proceedings of the International Conference on Computational Science (ICCS'03)*, volume 2659 of *Lecture Notes In Computer Science*, pages 750–759, Melbourne, Australia, June 2003. Springer-Verlag.

[36] M. Debbabi, M. Erhioui, L. Ketari, N. Tawbi, H. Yahyaoui, and S. Zhioua. Method Call Acceleration in Embedded Java Virtual Machines. In *Proceedings of the International Conference on Computational Science (ICCS'03)*, volume 2659 of *Lecture Notes in Computer Science*, pages 750–759, Melbourne, Australia, June 2003.

[37] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A Dynamic Compiler for Embedded Java Virtual Machines. In *Proceedings of the 3rd ACM Conference on Principles and Practice of Programming in Java (ACM PPPJ'04)*, pages 100–107, Las Vegas, USA, June 2004. ACM Press.

[38] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. A Synergy between Efficient Interpretation and Fast Selective Dynamic Compilation for the Acceleration of Embedded Java Virtual Machines. In *Proceedings of the 3rd ACM Conference on Principles and Practice of Programming in Java (ACM PPPJ'04)*, pages 108–115, Las Vegas, USA, June 2004. ACM Press.

[39] P. Dencker, K. Durre, and J. Heuft. Optimization of Parser Tables for Portable Compilers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(4):546–572, October 1984.

[40] L. Deutsch and A. Schiffman. Efficient Implementation of the Smalltalk-80 System. In *Proceedings of the 11th Symposium on Principles of Programming Languages (POPL'84)*, pages 297–302, Salt Lake City, UT, January 1984. ACM Press.

[41] K. Dixit and W. Bays. Frequently Asked Questions (FAQs) About the SPECjvm98 Benchmark. `http://www.specbench.org/osg/jvm98/qanda.html`, February 2001.

[42] K. Driesen. Selector Table Indexing and Sparse Arrays. In *Proceedings of the Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'93)*, Washington, DC, September 1993. ACM Press.

[43] S. Drossopoulou and S. Eisenbach. Is the Java Type System Sound. In *Proceedings of the 4th Workshop on Foundations of Object Oriented Languages (FOOL'97)*, pages 41–82, January 1997.

[44] S. Drossopoulou and S. Eisenbach. Describing the Semantics of Java and Proving Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 41–82. Springer-Verlag, June 1999.

[45] M. Ertl. A Portable Forth Engine. In *Proceedings of the European Forth Conference (EuroFORTH'93)*, Marienbad, Czech Republic, October 1993.

[46] Esmertec. `http://www.esmertec.com`.

[47] S. Freund and J. Mitchell. The Type System for Object Initialization in the Java Bytecode Language. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(6):1196–1250, November 1999.

[48] E. Gagnon and L. Hendren. Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences. In *Proceedings of the 12th International Conference on Compiler Construction (CC'03)*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184, Warsaw, Poland, April 2003. Springer-Verlag.

[49] P. Gastin and M. Mislove. A Truly Concurrent Semantics for a Simple Parallel Programming Language. In *Proceedings of the Annual Conference of the European Association for Computer Science Logic (CSL'99)*, volume 1683 of *LNCS*, pages 515–529, Madrid, Spain, September 1999. Springer-Verlag.

[50] P. Gastin and M. Mislove. A Truly Concurrent Semantics for a Process Algebra using Resource Pomsets. *Theoretical Computer Science (TCS)*, 281(1–2):369–421, June 2002.

[51] P. Gastin and A. Petit. *The Book of Traces*, chapter Infinite traces, pages 393–486. World Scientific, 1995.

[52] P. Gastin and D. Teodosiu. Resource Traces: A Domain for Processes sharing Exclusive Resources. *TCS*, 278:195–221, May 2002.

[53] P. Di Gianantonio, F. Honsell, and G. Plotkin. Uncountable Limits and the Lambda Calculus. *Nordic Journal of Computing*, 2(2):126–145, Spring 1995.

[54] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, MA, USA, 1985.

[55] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison-Wesley, CA, USA, 1996.

[56] C. Gunter and D. Scott. Semantic Domains. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Sematics*, pages 633–674. Elsevier Science Publishers, Amsterdam, The Netherlands, 1990.

[57] P. Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions On Programming Languages and Systems*, 19(4):557–567, July 1997.

[58] M. Hennessy. Acceptance Trees. *Journal of the ACM (JACM)*, 32(4):896–928, January 1985.

[59] U. Holzle, C. Chambers, and D. Ungar. Optimizing Dynamically–Typed Object-Oriented Languages With Polymorphic Inline Caches. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'91)*, volume 512 of *Lecture Notes in Computer Science*, Geneva, Switzerland, July 1991. Springer-Verlag.

[60] C. Hsieh, J. Gyllenhaal, and W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture*, pages 90–99, Paris, France, December 1996. IEEE Press.

[61] Bytecodes Inc. Just In Time Compilers. `http://www.bytecodes.com`.

[62] Insignia. `http://www.insignia.com`.

[63] Insilicon. `www.insilicon.com`.

[64] K. Ishizaki, T. Yasue, M. Kawahito, and H. Komatsu. A Study of Devirtualization Techniques for a Java Just-In-Time Compiler. In *Proceedings of the ACM-SIGPLAN Conference on Object-Oriented Programmimg Systems, Languages and Applications (OOPSLA'00)*, pages 294–310, Minneapolis, Minnesota, USA, October 2000. ACM Press.

[65] J. Katoen. *Quantitative and Qualitative Extensions of Event Structures.* PhD thesis, University of Twente, 1996.

[66] L. Ketari. *Techniques Dynamiques d'Optimisation de Plateformes Java Embarquées.* PhD thesis, Université Laval, 2006. Forthcoming.

[67] M. Kwiatkowska and G. Norman. Metric Denotational Semantics for PEPA. In *Proceedings of the 4th Workshop on Process Algebras and Performance Modelling (PAPM'96)*, pages 120–138, Torino, Italy, July 1996. CLUT.

[68] D. Lacey, N. Jones, E. Wyk, and C. Frederiksen. Proving Correctness of Compiler Optimizations by Temporal Logic. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'02)*, pages 283–294, Portland, OR, USA, January 2002.

[69] C. Laneve. A Type System for JVM Threads. *Theoretical Computer Science (TCS)*, 290(1):741–778, January 2003.

[70] S. Lerner, T. Millstein, and C. Chambers. Automatically Proving the Correctness of Compiler Optimizations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, pages 220–231, San Diego, California, USA, June 2003.

[71] S. Liang and D. Viswanathan. Comprehensive Profiling Support in the Java Virtual Machine. In *Proceedings of the 5th Conference on Object-Oriented Technologies and Systems (COOTS'99)*, pages 229–240, Berkeley, CA, USA, May 1999. USENIX Association.

[72] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Version.* Addison-Wesley, CA, USA, 1999.

[73] M. Lindwer. Java in Embedded Systems. *Xootic Magazine*, pages 14–24, May 2001.

[74] S. Majercik and M. Littman. Using Caching to Solve Larger Probabilistic Planning Problems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI'98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI'98)*, pages 954–960, Menlo Park, July 1998. AAAI Press.

[75] G. Manjunath and V. Krishnan. A Small Hybrid JIT for Embedded Systems. *SIGPLAN Notices*, 35(4):44–50, April 2000.

[76] F. Maruyama. OpenJIT 2: The Design and Implementation of Application Framework for JIT Compilers. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, Berkeley, CA, USA, April 2001. USENIX Association.

[77] Sun Microsystems. `http://www.sun.com`.

[78] Sun Microsystems. The Java HotSpot Performance Engine Architecture, April 1999. White paper.

[79] Sun Microsystems. Connected, Limited Device Configuration. Specification Version 1.0, Java 2 Platform Micro Edition, May 2000. White paper.

[80] Sun Microsystems. Java 2 Platform Micro Edition Technology for Creating Mobile Devices, May 2000. White paper.

[81] Sun Microsystems. KVM Porting Guide, September 2001. White paper.

[82] Sun Microsystems. MIDP APIs for Wireless Applications: A Brief Tour for Software Developers, February 2001. White paper.

[83] Sun Microsystems. The Jain APIs: Integrated Network APIs for the Java Platform, June 2001. White paper.

[84] Sun Microsystems. CLDC HotSpot Implementation Virtual Machine, 2002. White paper.

[85] Sun Microsystems. *Java 2 Platform, Standard Edition, Version 1.4.2 API Specification.* Sun Microsystems Inc., version 1.4.2 edition, April 2003.

[86] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Secaucus, NJ, USA, 1982.

[87] R. Milner. *Concurrency and Communication.* Prentice-Hall, 1989.

[88] M. Mislove. Denotational Models for Unbounded Nondeterminism. *Electronic Notes in Theoretical Computer Science*, 1, April 1995.

[89] M. Mislove, A. Roscoe, and S. Schneider. Fixed Points without Completeness. *Theoretical Computer Science (TCS)*, 138(2):273–314, February 1995.

[90] F. Morris. Advice on Structuring Compilers and Proving them Correct. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL'73)*, pages 144–152, Boston, Massachusetts, USA, October 1973. ACM Press.

[91] A. Muir. Zucotto – Embedding the KVM in Hardware. `http://www.microjava.com/jvm/hardware/native/zucotto2?content_id=725`.

[92] S. Muthukrishnan and M. Muller. Time and Space Efficient Method Lookup for Object-Oriented Programs. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 42–51, Atlanta, Georgia, January 1996. ACM Press.

[93] A. Mycroft, P. Degano, and C. Priami. Complexity as a Basis for Comparing Semantic Models of Concurrency. In *Proceedings of the Asian Computing Science Conference (ACSC'95)*, volume 1023, pages 141–155, Pathumthani, Tailand, December 1995. Springer-Verlag.

[94] T. Nipkow, D. Oheimb, and C. Pusch. μJava: Embedding a programming language in a theorem prover. In *Proceedings of the Summer School on Foundations of Secure Computation*, volume 175 of *NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[95] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[96] M. Nunez, D. de Frutos, and L. Llana. Acceptance Trees for Probabilistic Processes. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, pages 249–263, Philadelphia, PA, August 1995.

[97] D. Oheimb and T. Nipkow. Machine-Checking the Java Specification: Proving Type Safety. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 119–156. Springer-Verlag, June 1999.

[98] S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. Pvs system guide. Technical report, Computer Science Laboratory, SRI International, 1999.

[99] M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Monterey, California, April 2001. USENIX Association.

[100] L. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[101] I. Piumarta and F. Riccardi. Optimizing Direct-Threaded Code by Selective Inlining. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*, pages 291–300, Montreal, Canada, June 1998. ACM Press.

[102] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, Computer Science Department, Aarhus University, 1981.

[103] G. Plotkin. An Operational Semantics for CSP. In *Proceedings of the IFIP TC 2-Working Conference on Formal Description of Programming Concepts*, pages 199–225, Amsterdam, Netherland, 1983.

[104] M. Poletto and V. Sarkar. Linear Scan Register Allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, September 1999.

[105] V. Pratt. Modelling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, May 1986.

[106] T. Proebsting, G. Townsend, P. Bridges, J.Hartman, T. Newsham, and S. Watterson. Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (COOTS'97)*, pages 41–54, Berkeley, USA, June 1997. USENIX Association.

[107] Z. Qian. A Formal Specification of Java Virtual Machine Instructions for Objects, Methods and Subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 271–312. Springer-Verlag, June 1999.

[108] C. Quigley. *A Programming Logic for Java Bytecode Programs.* PhD thesis, University of Glasgow, 2004.

[109] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[110] R. Radhakrishnan, N. Vijaykrishnan, L. Kurian John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, 50(2):131–146, February 2001.

[111] A. Roscoe and G. Barrett. Unbounded Nondeterminism in CSP. In *Proceedings of the 5th International Conference on Mathematical Foundations of Programming Semantics (MFPS'89)*, volume 442, pages 160–193, Tulane University, New Orleans, Louisiana, USA, March 1989. Springer-Verlag.

[112] V. Sassone, M. Nielsen, and G. Winskel. Models for Concurrency: Towards a Classification. *Theoretical Computer Science (TCS)*, 170(1–2):297–348, January 1996.

[113] K. Schmid. Esmertec's Jbed Micro Edition CLDC and Jbed Profile for MIDP, Spring 2002. White paper.

[114] B. Shannon. *Java 2 Platform Enterprise Edition, Version 1.3.* Sun Microsystems Inc., version 1.3 edition, July 2001.

[115] N. Shaylor. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium (JVM'02)*, pages 119–126, San Francisco, CA, USA, August 2002. USENIX Association.

[116] K. Shudo. Performance Comparison of JITs. `http://www.shudo.net/jit/perf/`, January 2002.

[117] I. Siveroni. *Correctness of Analysis-based Program Transformations of Functional Programming Languages.* PhD thesis, College of Computer Science, Norhteastern University, 2002.

[118] Pendragon Software. CaffeineMark. `http://www.pendragon-software.com`, 1996.

[119] V. Sreedhar, G. Gao, and Y. Lee. Identifying Loops using DJ Graphs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(6):649–658, November 1996.

[120] V. Sreedhar, G. Gao, and Y. Lee. Identifying Loops in almost Linear Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 21(2):175–188, March 1999.

[121] B. Steensgaard. Sequentializing Program Dependence Graphs for Irreducible Programs. Technical Report MSR-TR-93-14, Microsoft Research, Redmont, Wash., October 1993.

[122] K. Stephenson. Compiler Correctness using Algebraic Operational Semantics. Technical Report CSR 1-97, University of Wales Swansea, 1997.

[123] M. Strecker. Formal Verification of a Java Compiler in Isabelle. In *Proceedings of the Conference on Automated Deduction (CADE'02)*, volume 2392 of *Lecture Notes in Computer Science*, pages 63–77, Copenhagen, Denmark, July 2002. Springer-Verlag.

[124] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, February 2000.

[125] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, January 2000.

[126] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-In-Time Compiler. *ACM SIGPLAN Notices*, 36(11):180–195, November 2001.

[127] D. Syme. Proving Java Type Soundness. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523, pages 83–118. Springer-Verlag, June 1999.

[128] R. Tarjan. Testing Flow Graph Reducibility. *Journal of Computer and System Sciences*, 9:355–365, August 1974.

[129] Coq Development Team. The Coq Proof Assistant Reference Manual, Version 8.0. Technical report, INRIA, 2004.

[130] J. Thatcher, E. Wagner, and J. Wright. More Advice on Structuring Compilers and Proving them Correct. *Theoretical Computer Science (TCS)*, 15:223–249, 1981.

[131] S. Vijay, H. Laurie, R. Chrislain, V. Raja, L. Patrick, G. Etienne, and G. Charles. Practical Virtual Method Call Resolution for Java. In *Proceedings of the ACM-SIGPLAN Conference on Object-Oriented Programmimg Systems, Languages and*

Applications (OOPSLA'00)), pages 264–280, Minneapolis, Minnesota, USA, October 2000. ACM Press.

[132] M. Walicki and S. Meldal. Algebraic Approaches to Nondeterminism-An Overview. *ACM Computing Surveys*, 29(1):30–81, 1997.

[133] M. Wand. Compiler Correctness for Parallel Languages. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 120–134, La Jolla, CA, USA, June 1995. ACM Press.

[134] M. Weiss, F. Ferrière, B. Delsart, C. Fabre, F. Hirsch, E. Johnson, V. Joloboff, F. Roy, F. Siebert, and X. Spengler. TurboJ, A Java Bytecode-to-Native Compiler. In *Proceedings of the ACM SIGPLAN Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'98)*, volume 1474, pages 119–130, Montreal, Canada, June 1998. Springer-Verlag.

[135] G. Winskel. *Events in Computation.* PhD thesis, University of Edinburgh, 1980.

[136] G. Winskel. Synchronization Trees. *Theoretical Computer Science (TCS)*, 34:32–82, 1984.

[137] G. Winskel and M. Nielsen. *Handbook of Logic in Computer Science*, volume 4, chapter Models for Concurrency. Clarendon Press, 1995.

[138] P. Wolper and P. Godefroid. Partial-Order Methods for Temporal Verification. In *Proceedings of the 4^{th} International Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 233–246, Berlin, Heidelberg, August 1993. Springer-Verlag.

[139] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proceedings of the IEEE Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 128–138, California, USA, October 1999. IEEE Press.

[140] B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y.C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'99)*, pages 128–138, Newport Beach, California, October 1999. IEEE Press.

[141] F. Yellin. Inside the K Virtual Machine (KVM). In *JavaOne 2000*, Moscone Convention Center, San Francisco, California, June 2000.