Dottorato di Ricerca in Informatica

Università di Bologna, Padova, Venezia

Formalization, Analysis and Prototyping of Mobile Code Systems

Cecilia Mascolo

January 2001

Coordinator:

Prof. Özalp Babaoğlu

Tutor:

Prof. Paolo Ciancarini

A mia nonna Onesta
al suo impareggiabile desiderio di imparare
To my grandmother Onesta
to her unmatched desire for learning

Abstract

In the past few years dynamic and reconfigurable systems have evolved and new strategy and paradigms for the development of applications have been devised. In this thesis we study mobile code based systems focusing on the importance of formalization and investigation of the potential of code mobility. Mobile code paradigms have been used in different systems, however, as most of these are Java based, the potential of code mobility are some-how lost behind the Java language capabilities, and design choices related to mobility have been conditioned by implementation choices.

In this thesis we reason on code mobility systems at the design level in order to investigate novel powerful approaches. This thesis is composed of different parts. We first introduce a coordination based language and a model checker to reason on formalization of mobile code based systems with automatic analysis. Properties of mobile agents, of their interaction and behavior may be formally expressed and verified against the system specification.

Then, in order to express code mobility potential and to formalize the basic constructs for code migration, we describe a formal language for the specification of very fine-grained mobility. Every line of code, and every variable declaration can be mobile, giving a very high flexibility in the range of application. A prototype of this model implemented in Java is also presented to validate the implementability of the model.

Finally, we show a possible incarnation of the fine-grained mobility approach based on XML. The approach allows XML documents to be updated cutting, extending, or replacing parts of the tree structure of the document. We exploit this idea to incrementally update remote code. The approach can be used in different domains; we describe possible applications in graphic user interface management, document consistency checking and management of application on thin clients like personal digital assistants (PDAs).

Contents

List of Tables **xiii**

List of Figures **xv**

Introduction **1**

1 Code Mobility: Technologies and Formalisms **5**

1.1 Mobile Code Technologies 6

1.2 Mobile Code Formalisms 10

1.3 Summary 13

I Formalization and Analysis of Mobile Code **15**

2 PoliS: a Coordination Approach to Formalization **17**

2.1 Overview of PoliS 17

2.2 Abstract Syntax and Operational Semantics for PoliS 22

2.3 Specification in PoliS 27

2.4 PoliS and Software Architectures 31

3 The PoliS Model Checker **35**

3.1 Model Checking a Coordination Model 35

3.1.1 The PoliS Graph Construction 36

3.1.2 The PoliS Temporal Logic 39

3.1.3 The PoliS Model Checker 41

3.2 Analysis of the Invoice System 42
3.3 Model Checking Software Architectures 45

4 PoliS Specification and Analysis of Mobile Code Systems 49
4.1 Specification of an Architecture with Mobile Agents 53
4.1.1 The Meeting Scheduler System: an Informal Description 53
4.1.2 A Specification including Mobile Agents 54
4.1.3 Analysis of the Meeting Scheduler System 59

5 MobiS: an Enhancement of PoliS 63
5.1 MobiS 63
5.2 The semantics and the difference with PoliS 65
5.3 Using MobiS for Agent Mobility across a Network 67
5.4 Specification of Architectural Styles for Mobility 72
5.5 Application of the Styles to the Architecture of a Mobile System 77

6 Summary 81

II Fine-Grained Approach to Mobility: Formalization and Prototyping 85

7 A Fine-Grained Model 87
7.1 Model Overview 88
7.2 Mobile UNITY 90
7.3 Reinterpretation of the Mobile UNITY syntax 93
7.4 Mobility Constructs 97
7.5 Formal Semantics 103
7.5.1 Scoping Rules 105
7.5.2 Statement Scheduling 107
7.5.3 Mobility Constructs 107

7.5.4 Creation Predicates 110

8 An Enhanced Fine-Grained Model **113**

8.1 The Enhancement with Scoping 113

9 Lilliput: a Fine-Grain Mobility Prototype **119**

9.1 The LILLIPUT System 120

9.2 The Architecture of the System 126

9.3 Implementation Details 128

9.3.1 The Engine 130

9.3.2 The Interpreter 131

9.3.3 The Handler 132

9.4 Discussion and Future Work 133

10 Summary **135**

III Implementing Incremental Code Mobility **139**

11 Active Documents **141**

11.1 Managing complex documents over the Internet 141

11.2 Creating Z specifications 143

11.2.1 Hypertext and Z specifications 144

11.2.2 The advantages of markup languages 146

11.3 Displets and markup languages 147

11.3.1 Applying displets to XML documents 149

11.4 The Rendering Engine 152

11.5 The Z browser 155

12 Incremental Code Mobility and Applications **161**
12.1 Overview of XML and Logical Mobility 162
12.2 Specifying Incremental Code Mobility with XML 164
12.3 Implementation of the Approach 168
12.3.1 Interpreter Implementation 168
12.3.2 Code Mobility 170
12.3.3 Incremental Code Mobility 172
12.4 Applications 173
12.4.1 User Interface Engines 174
12.4.2 Application Management on Mobile PDAs 176
12.4.3 Consistency Management 177
12.5 Evaluation 179

13 Summary **181**

Conclusions **183**

A The Lilliput Input Grammar **187**

B The Lilliput API **191**

References **203**

List of Tables

2.1 Specification of a Client-Server System in PoliS. 21
2.2 PoliS Abstract Syntax. 23
2.3 Structured Operational Semantics Rules and Axioms. 24
2.4 Classification of PoliS Rules Macro. 25
2.5 Precondition predicates. 26
2.6 PoliS specification of Case1 28
2.7 PoliS specification of Case2 30

3.1 Specification of a simple component. 37
3.2 Specification for *StartContext*. 38
3.3 PTL Syntax. 40

4.1 Specification of a Client-Server System with a Mobile Agent: first part. . . 50
4.2 Specification of a Client-Server System with a Mobile Agent: second part. . 51
4.3 The Meeting Scheduler: the Main Space. 56
4.4 The Meeting Scheduler: the Participant Space. 56
4.5 The Meeting Scheduler: the Agent Space. 57

5.1 Specification of the Client component 64
5.2 MobiS Abstract Syntax. 67
5.3 MobiS Structured Operational Semantics Rules and Axioms. 67
5.4 Classification of PoliS Rules Macro. 68
5.5 Precondition predicates. 68
5.6 Specification of the Network with Agents System: the WAN and LAN spaces. 69

5.7 Specification of the Network with Agents System: the Host Space. 70
5.8 Specification of the Network with Agents System: the Agent space. 71
5.9 MobiS specification of Code Paradigm 74
5.10 MobiS specification of the Data Paradigm. 75
5.11 MobiS specification of the Ambient style 76
5.12 Specification of the Purchasing Architecture in the Ambient Style 78

List of Figures

1.1 The object shipping process in Java. 7
1.2 The object shipping process in Java. 8

2.1 PoliS nested spaces (a) and the corresponding tree interpretation (b). . . . 18
2.2 The scope of a rule. 19
2.3 Creation (a) and Termination (b) of spaces. 20
2.4 Structure of the Invoicing System (Case1). 28
2.5 Strcuture of the Invoicing System (Case2). 29
2.6 Architecture of a Server with two Handlers. 32
2.7 Architecture of a Client-Server System with a Layered Router. 33

3.1 Graph for the simple space Component. 37
3.2 Representation of configurations. 39

4.1 A simple Client-Server system with a Mobile Agent. 50
4.2 Agent Mobility in PoliS. 53
4.3 Agents System Architecture. 55
4.4 Agent performing update. 59

5.1 The movement of a space. 66
5.2 The Network. 68

7.1 Processes, units, and scoping rules. 89
7.2 A Mobile UNITY system :distributed computation of the minimal value of x. 91
7.3 Fine-grained restructuring of the *ElectionAgent* System. 94

7.4 A node in the leader election solution with processes. 97
7.5 Leader Election in Mobile UNITY extended with fine-grained mobililty constructs. 99
7.6 Leader election with mobile code. 100
7.7 Specification of the functions find and exists. 104
7.8 Bindings among units using variable sharing and statement inhibition. . . . 106
7.9 Mapping mobility constructs to Mobile UNITY statements. 109
7.10 Modeling the actions of the run-time support. 110
7.11 Migrating components. 111
7.12 Constructs for the instantiation of components. 111

8.1 The structure of a host in the enhanced model (a) and the corresponding tree topology (b). 114
8.2 Scoping in the enhanced model: the dashed circles represent data units. . . 115
8.3 Referencing in the hierarchical model 116
8.4 Migrating components: enhanced model. 117
8.5 Updated functions return values. 117

9.1 Translation of the Input Document. 120
9.2 An example of system as input in LILLIPUT. 121
9.3 The Java representation of the data unit for variable x. 122
9.4 The Java representation of the code unit for statement migrate. 122
9.5 The Java representation of the **Components** section. 124
9.6 A host of the LILLIPUT system. 125
9.7 The initial system configuration. 126
9.8 The LILLIPUT engine interface. 127
9.9 The LILLIPUT interpreter thread main cycle. 128
9.10 The Class Diagram of LILLIPUT. 129

11.1 The general structure of the DispletManager applet 149
11.2 Rendering a simple displet 152

11.3 The inheritance structure of the module library 154
11.4 Visualization on the WWW of a Z schema 159

12.1 The DTD for Karel's Instruction Set. 164
12.2 An XML program for Karel. 165
12.3 The actions of the robot. 166
12.4 XML Code Increment. 166
12.5 The incremental change to Karel's behaviour. 168
12.6 XPointer Address for Increment. 168
12.7 Translating XML program into an AST. 169
12.8 Abstract Syntax Tree for Karel's Program. 170
12.9 Traversing the AST during interpretation. 171
12.10Migration of XML program to remote interpreter. 171
12.11Remote Method Invocation for Karel. 172
12.12The migration of the increment XML file to the robot site. 172
12.13Result of incremental code update on AST. 173
12.14Evaluating XPointer Expression 174
12.15A Consistency Rule in XML Format. 177
12.16Consistency management architecture. 178

Introduction

The increasing popularity of Java and the spread of Web-based technologies are contributing to a growing interest in dynamic and reconfigurable distributed systems. The ability to relocate code over networks of workstations, on an Internet scale, yields flexibility in the design of new applications and shows new possible paths to be followed in the development of the future systems.

Code mobility is viewed by many as a key element of a class of novel design strategies which no longer assume that all the resources needed to accomplish a task are known in advance and available at the start of the program execution. Know-how and resources are searched for across the networks and brought together to bear on a problem as needed. Often the program itself (or portions thereof) travels across the network in search of resources. While research has been done in the past on operating systems that provide support for process migration, mobile code languages offer a variety of constructs supporting the movement of code across networks. Java [Sun95], Tcl [Gra95], and derivatives support the movement of architecture-independent code that can be shipped across the network and interpreted at execution time. Obliq [Car95] permits the movement of code along with the reference to resources it needs to carry out its functions. Telescript [Whi96] is representative of a class of languages in which fully encapsulated program units called agents migrate from site to site. Location, movement, unit of mobility, and resource access are concepts present in all mobile code languages. Differentiating factors have to do with the precise definitions assigned to these concepts and the operations available in the language [FPV98].

Language design efforts are complemented by the development of formal models. Their

main purpose is to gain a better understanding of fundamental issues facing mobile computations. Of course, such models are expected to play an important role in the formulation of precise semantics for mobile code languages and constructs, to serve as a source of inspiration for novel language constructs, and to uncover likely theoretical limitations.

Motivation and Contribution

The aim of this work is to achieve a deep insight in mobile code based systems and to investigate some possible developments in this field. Mobile code has been exploited in different technologies, however, as most of them are Java-based, the focus on the potential of mobile code is somehow lost behind the capabilities of Java, and mobility design choices have been conditioned by implementation constrains. The aim of the thesis is to abstract from the current technologies investigating the real power of the migration of code across the network. In particular, the idea is to use formalisms to achieve understanding of mobile code systems. In this context, automatic analysis of specification can be useful in finding conceptual mistakes in systems specifications, and prototyping of new mobile code based paradigms can give insight into their potentials.

We will describe how a coordination language can be used to specify the dynamics of mobile systems. On top of the language, a model checker will be used to analyze properties of mobile code in an automatic way. We will then introduce a more programming-oriented formalism to study the issues related with the granularity of the unit of mobility and its decoupling from the unit of execution. In this approach we describe a prototype of the language to give details of the implementability of the idea. The formal study succeeded in isolating interesting future trends for mobile code. The last part of the thesis shows how recently developed technologies and languages happen to incarnate interesting fine-grained characteristics that other existing mobile code languages lack and that can be applied in domains such as distributed application management and mobile computing settings.

Outline of the Thesis

Chapter 1 recalls some background concepts related to code mobility. The chapter is intended as an introduction to the basic notions used in the rest of the thesis. We also describe some related work in terms of both existing mobile code technologies and formal languages. The thesis is composed of three main parts.

In **Part I** we show how mobile code specifications can be formalized and automatically analyzed using a coordination languages and a model checker. The specification and the analysis of software architectures are also described and mobility architectural styles are specified. In particular, **Chapter 2** introduces PoliS, a coordination based languages that has already been used for the description of complex systems. PoliS has characteristics of flexibility that permit the formalization of complex systems. In particular, we show how software architectures can be specified in PoliS. **Chapter 3** describes the logic (PTL) and the model checker for PoliS developed at the University of Bologna. In this chapter we use the model checker to prove properties on PoliS specification of systems. Examples of analysis of software architectures are shown. **Chapter 4** introduces the use of PoliS for the specification of mobile code based systems. The model checker is used to perform analysis of mobile systems. **Chapter 5** contains the description of MobiS, an enhancement of the PoliS language that is able to formalize mobile agents as first class elements in the language. **Chapter 6** contains a summary of the part.

In **Part II** we show how the granularity of mobility can be refined until a very fine-grained level. **Chapter 7** describes a more programming language oriented formalism, an enhancement of Mobile UNITY [MR98] for specification of mobile code based systems. In this model we adopt the view that every line of code and every variable can be mobile. The unit of mobility is then decoupled from the unit of execution and dynamic system reconfiguration is possible at a very fine-grained level. **Chapter 8** contains an enhancement of the model described in Chapter 7, with nested processes, while **Chapter 9** shows a design and a Java prototype of the model presented in the previous chapter, in order to highlight the feasibility and the implementability of the approach. **Chapter 10** contains a summary of the part.

In **Part III** we show how to use XML [BPSM98a] for incremental code mobility. We describe some applications in different domains. In **Chapter 11** we describe the use of XML (i.e., the EXtensible Mark-up Language) plus Java class loading for display of formal notation documents on the Internet. As follow-up of this work and of work in Chapter 7, **Chapter 12** shows the incremental code mobility approach based on XML and related technologies. The approach can be applied to different application domains, and we give some examples. **Chapter 13** contains a summary of the part.

The **Conclusions** chapter contains the summary of the work and a list of possible

developments. **Appendix A** contains the grammar for the input notation of the prototype shown in Chapter 9, and **Appendix B** its Application Programming Interface.

Related Publications

Part of this thesis has been taken from published papers, in particular in [Mas99b] an outline of the thesis is presented. Chapter 2 and Chapter 3 on the PoliS language and specification of systems and software architectures are an evolution of different papers; in [CM98c] we use the language and the model checker to analyze some invoicing systems: the invoice system was the case study offered in the International Workshop on Comparing System Specification Techniques held in Nantes, France in 1998. [CM99, CM98a] describe the use of PoliS and the checker for software architectures. The papers [CFM98], and [CFM00] introduce the use of PoliS for the specification of mobile code based systems. Chapter 5 refines a paper presented in [Mas99a], describing the MobiS language, an evolution of PoliS allowing first class formalization of mobile agents. MobiS has also been used to specify software architectures with mobile components in [CM98b]. Chapter 7, 8 and 9, where the fine-grained model and prototype is presented, refine [MPR99], and [MPR00]. Chapter 11 extends [CMV98] and [CVM99], while Chapter 12 refines a paper published in [EMF00] and in [MEF00].

During my Ph.D. I also published other papers, that, for sake of brevity, are not part of this thesis. In [CCM96], and [CCM97] semantics for the Z language [Spi92] based on the Chemical Abstract Machine is presented, and an animator of Z specification based on the semantics given is used to test specifications. In the same context we used the approach to analyze dynamics of systems [CM96], and to describe architectural styles [CM97].

In [CM98d] an approach to the use of formal method for teaching software engineering is presented: the approach focuses on the use of tools to improve the generation of software specification and design documents.

Chapter 1

Code Mobility: Technologies and Formalisms

Logical mobility is not a new concept. Data transfer have been used to exchange or distribute information among different people on a network. For a long time data have been transmitted across network using e-mail and ftp protocols. With the spread of the World Wide Web, HTML documents can be sent over the Internet using the http protocol.

The increasing popularity of Java is contributing to a growing interest in dynamic and reconfigurable systems [MDEK95]. In particular, the ability to relocate not only data but also code over networks of workstations, on an Internet scale, yields flexibility in the design of new applications.

From this starting point mobile code evolved, and more complicated mobile code paradigms have been isolated [FPV98]: code on demand, remote evaluation, and mobile agents. In particular it became feasible to send objects, with their status and their code, from a location to another. *Object serialization* is the mechanism used for the transmission across the network of objects in Java. When an object has some sort of autonomous behavior and proactivity it is called a *mobile agent* [WPM99]. Java based mobile code technologies developed very rapidly and many different systems based on mobile agents have been built in the last years. Mobile agents are able to travel carrying their own status and code form location to location, following a itinerary or following some sort of pattern. We will describe most of them in Section 1.1.

On the theoretical front, the growth of languages able to express mobile code characteristics did not have the same power of growth than technologies. However some interesting approaches have been adopted in this direction.

Language design efforts are complemented by the development of formal models. Their

main purpose is to gain a better understanding of fundamental issues facing mobile computations. Of course, such models are expected to play an important role in the formulation of precise semantics for mobile code languages and constructs, to serve as a source of inspiration for novel language constructs, and to uncover likely theoretical limitations. Basic differences in mathematical foundation, underlying philosophy, and technical objectives led to models very diverse in flavor. In Section 1.2 we give an outline of the existing formalisms used for specifying code mobility systems.

1.1 Mobile Code Technologies

Many different technologies have been developed in the recent years based on code mobility concepts. In this section we will introduce some of the most common technologies.

In [FPV98] a classification of mobile code technologies is given. The paper distinguish between weak and strong mobility. Weak mobility is the ability to move code and the status of an object, and it is the most common kind of mobility provided by the systems we are going to describe. Strong mobility is the ability to move not only the code and the state of an object but also the execution status, that is, the program counter and the registers of the executing object. Strong mobility is more complicated to achieve as it offers a higher level of complexity.

Java

The Java programming language [Sun95] is an object-oriented language that allows classes and objects to be serialized and written into a stream of bytes in order to be transmitted. In this section we will only focus on mobility related aspects and not on the all language as it would go beyond the scope of this thesis.

Java objects are instances of Java classes. Each class of object that needs to be transferred over the network (i.e., serialized) needs to "implement" the `serializable` interface. Serialization of object is a Java mechanism allowing the status of an object to be written into a byte-stream. Once an object is serialized it may be shipped to remote locations. The classes on which the object relies are however not serialized in the same byte stream and therefore not implicitly transmitted with the object.

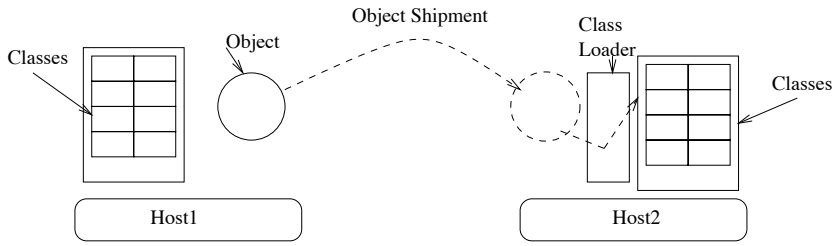

Figure 1.1: The object shipping process in Java.

The Java class loader is responsible for the loading of classes of objects. The class loader uses the `CLASSPATH` environment variable to know where to retrieve a class for an object. Per se, the Java Class Loader throws an exception every time the needed class cannot be found in the specified directories of the `CLASSPATH` . However, the class loader can be overridden and specific class loading policies can be used to load classes, even remotely.

Figure 1.1 shows the transmission of an object and the class loading happening on the remote site. The default class loader looks into the local `CLASSPATH` environment variable to retrieve the class for the received object.

The Java API also provides a networking package for communication through sockets, that is often used for migration purposes in the mobile code technologies based on Java for mobile agents [WPM99].

Java provides weak mobility, as serialization of threads in Java is not possible. This is one of the reasons why most of the common mobile agents systems, which are developed on Java, are based on weak mobility.

Java Applets

Java can be used together with Web browsers in order to achieve code mobility on the Internet. This was in fact the first use of Java for code mobility. Figure 1.2 shows the idea. Java applets are pointed from HTML documents. After an HTML page containing a Java applet is downloaded by a client browser, the browser class loader has to retrieve the Java code for the applet from the location where the applet comes from, fetch it, and load it.

Java applets are probably the most well known example of code mobility. Since Java

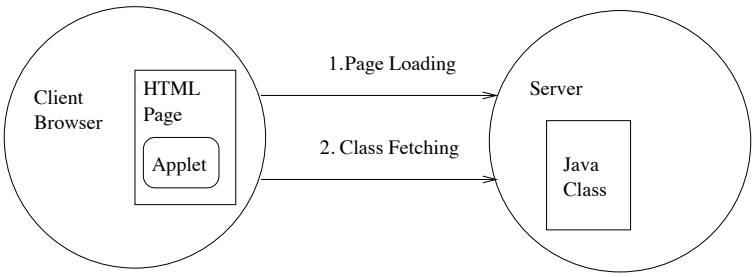

Figure 1.2: The object shipping process in Java.

applets began to be used over the Web, code mobility issues have pop up, and research began to investigate on this topic, and realizing that Java code mobility could actually be Web independent.

Java RMI

Java RMI [RMI98] is part of the Java API, however it deserves particular attention in a mobile code context. RMI stands for Remote Method Invocation; the RMI package allows remote invocation of methods in distributed objects. The client invoking the remote method can introduce parameters to the calls. The calls to the methods can be done *by value* or *by reference*. In calls by reference the object a reference to the object is passed during the call. The object needs to implement the `Remote` interface in order to be passed by reference. In calls by value the objects passed as parameters are copied remotely. In a mobility perspectives, calls by value allow to migrate objects (by copying and deleting them) from one location to another. A method can be invoked remotely only if extending the interface `Remote` , and an object can be passed as a parameter by value only if its class extends the `serializable` interface (and not `remote`). To allow remote invocations, stubs and skeletons of the object on which the invocation needs to take place are provided, on the client and server side, respectively.

Aglets

Aglets [LO98] is probably the most well known system for mobile agents. The Aglets system is developed on top of Java. It exploits serialization and networking with sockets for the mobility of agents. An aglet is an object able to move across the network. In particular an aglet moves from "place" to "place". A place is a context in which aglets

execute. Whenever an aglet needs to be moved, it is suspended, the Java serialization mechanism is used to write the aglet into a byte-stream, and then the aglet is transferred. On the other side, the aglet is received, deserialized and its execution is resumed with a new thread of execution. Aglets, as based on Java, provides agents mobility at the weak level, i.e., no mobility of the execution state. Whenever an aglet is received on a new host the classes used by the agent are can be retrieved on the new host itself. The classes could also be transferred together with the aglet itself, or retrieved from a remote server that is supposed to store the class for that purpose. Aglets can communicate with each other on the same place or among places in peer-to-peer or broadcast fashion. IBM's Aglets can be downloaded at the web site: `www.trl.ibm.co.jp/aglets` .

Voyager

Voyager [Obj97] is an object request broker (ORB) that also provides mobile agents facilities. A voyager is a location on which different agents can live. Agents in the same voyager can communicate and exchange data. An agent can migrate from one voyager to another, proactively.

Objects in voyagers have proxies that enable, like skeletons and stubs in Java RMI, remote invocations. Voyager is again implemented in Java, and therefore provides weak code mobility like Aglets.

A part from these mobility features Voyager is a middleware like CORBA [OMG95] and COM [Gri97], providing a transparent and reliable communication layer on which applications can be developed.

μ-Code

μ-CODE [Pic98] will be used in Chapter 9 for the implementation of a fine-grained mobility prototype. We give here a brief description of its features.

μ-CODE is a Java based system providing weak mobility; it implements agents mobility as well as code mobility at a class level. Unlike in other systems where classes are fetched only following an agent migration, and in order to make the agent able to execute, in μ-CODE migration of classes is an invokable operation of the system API. Groups, i.e. bags, of objects, agents, and classes can be shipped and fetched across the network allowing a high level of flexibility. Application developers can then use groups to send things across

the network and the dimension of the group and the nature of the entities depends on the application requirements. The granularity of mobility is then system independent and can go from a class to an agent carrying its status.

μ-CODE relies on Java sockets for the implementation of the communication even if its API hides these detail from the programmer. μ-CODE is designed to be flexible, extensible and light-weight. It is composed of less than a thousand lines of code that generate less than 40kbytes of byte-code.

Other Technologies

Many other technologies have been developed exploiting mobile code, however we described the most significant and the ones that we are going to mention in the following chapters. Some other example of mobile code systems need however to be at least referenced. There are a few interesting approaches to mobility that are not Java based: Emerald [LHM88], Telescript [Whi96] and Agent-Tcl [Gra95]. In particular Telescript allows strong mobility, that is, agents can move from place to place restarting execution from the exact point they left it on the previous place. Strong mobility has also been achieved through a Java approach in [Fru98], however the approach is quite complicated and most of the Java based systems do not apply it. Details about other mobile agents systems are given in [WPM99]. In [RPZ97],[RH98],[LM99], and [MK00] the proceedings of a newly established mobile agents conference contains the description of other mobile code systems.

1.2 Mobile Code Formalisms

Many formalisms have been used or developed to be able to express mobile code based systems. In particular process algebra is one of the most successfully used theory in this respect. In this section we will briefly describe some of them, as they will be compared to the formalisms presented in this thesis later on. Formalisms have the important role to abstract from technologies and specify characteristics of the behavior of systems, providing useful background for investigation and analysis.

π-calculus and Derived Formalisms

The π-calculus [Mil99] is probably the first formal language being exploited for the specification of mobility aspects. In π-calculus, as in all the process algebra based models the *process* is the unit of mobility and the unit of execution. Processes can be sent around, they can perform computations, and communicate with each other . π-calculus [Mil99] is based on the notion of *channels*. Processes can move along channels. Code mobility is exactly represented as migration of processes. π-calculus has also been extended in several direction in order to overcome for instance the lack of notion of location (Join calculus [FGL$^+$96]), or to add asynchronous mechanisms to the model [Ama97].

Klaim

Klaim [NFP98] is a process algebra based language that exploits coordination primitives à la Linda [CG92] to express the notion of location. Klaim has been implemented in Java and the resulting API has been called KLAVA. Klaim allows the formalization of different paradigms of mobility, from the fetching and shipping of code to mobile agents moving with their contexts.

Obliq

Obliq [Car95] is a lexically-scoped interpreted language. The Obliq environment consists of sites, i.e., addresses spaces containing locations. Code and objects references can be moved from site to site. Objects are not allowed to migrate, however they can be cloned and put on remote sites. In addition it is possible to perform aliasing of the original objects to redirect method invocations to the cloned objects.

Ambient Calculus

Mobile Ambients [CG00] is a process algebra based language allowing computation to move with their contexts. The language is quite powerful, allowing the moving of computations together with their environment. The language relies on the notion of *ambient*. An ambient can contain other ambients, and it defines the scope of computations contained in it. On top of the ambient calculus some security mechanisms based on capabilities, i.e., access

rights, and types have been defines. Every ambient has a name that allows capabilities to be defined on it in order to constrain access rights.

Mobile UNITY

Mobile UNITY [MR98] will be used and refined in Chapter 7. We will give a general outline here to leave the description of the details when we need them. Mobile UNITY is a state based language based on UNITY [CM88]. Mobile UNITY allows the definition of programs, and of their mobile behavior, assigning to each program a location variable. The unit of mobility, as well as the unit of execution and of definition in Mobile UNITY is the program. Users write programs that can be instanced an migrated over the network re-assigning the location variable. The **Interactions** section of a Mobile UNITY document defines the interactions and the movements of the components. The **Components** section defines the instantiations of programs that are going to exist in the system. The set of components is then fixed.

Mobile UNITY has a well-defined proof system based on temporal logic that allows the specification and verification of properties on mobile systems.

Other Formalisms

As in the previous section for technologies, in this section we only described some of the relevant approaches to formalization of mobile code systems. Other approaches have been developed, and some of them deserve to be at least mentioned, like [VC99], [PS99] and [WF98], and the paper in [SMT98] presents a survey of existing formal approaches to code mobility.

The research on distributed and reconfigurable systems and evolving architectures is not focused on mobility but has similarities and common issues, like rapid evolution and constant changes. Some relevant work on this topic deserves to be mentioned. In [IW95] the Chemical Abstract Machine [BB92] (CHAM), a formal model based a chemical metaphor, is used to express software architectures. The CHAM allows to express environment as chemical solution of molecules that evolve based on a set global rules. The chemical metaphor allows the specification of modular components and of the interactions among them. In [GKC99] a process algebra based approach to dynamic software architecture is presented. A model checker is used to investigate behavioral properties on the

specifications such as liveness and safety. In [WF99] and [OT98] different approaches with specific focus on reconfiguration and dynamics analysis are presented.

1.3 Summary

In this chapter we have discussed some relevant related work. The chapter focused on existing mobile code technologies and formalisms. Technologies have been developed very quickly in this field, exploiting some new ideas. However the role played by formalisms is very important as they permit to discover, at an abstract level, some possible future developments in the field. In this thesis we combine formal specification approaches and analysis to implementation and prototyping in order to have on one hand the abstract shape of the ideas, and on the other the validation of them with respect to the real applicability.

In the following we will often mention the described related work presented in this chapter and compare it with our approach in order to clarify the main contribution of this work.

Part I

Formalization and Analysis of Mobile Code

Chapter 2

PoliS: a Coordination Approach to Formalization

This chapter describes the language, named PoliS, that we use for the specification of mobile code systems. PoliS is a coordination language based on a multiple tuple-spaces model. PoliS can be used to specify and analyze systems based on logical mobility: code mobility is represented as a first class concept. PoliS [CMP98] has already been used to specify the architectures of complex systems in the past. The coordination media in PoliS are multiple tuple spaces, which offer a natural basis for describing mobile entities and their dynamic reconfiguration. The pattern matching mechanism adopted to access the tuple spaces helps in abstracting away from low level addressing issues. Code can be explicitly moved from one PoliS space to another, duplicated, and eliminated.

In Chapter 3 we show how we can use a model checking approach to analyze properties on PoliS specifications, and in Chapter 4 we describe how we applied the approach to mobile code based systems.

2.1 Overview of PoliS

PoliS is a coordination language whose coordination media are nested tuple spaces [CMP98]. A tuple space, or *space* for short, includes as coordinables both tuples and other spaces. PoliS specifications are modular and hierarchically structured: a PoliS specification denotes a tree of nested spaces that dynamically evolves over time. Figure 2.1.a shows a structure of nested spaces (i.e., the nested circles); Figure 2.1.b shows the corresponding tree whose nodes are the spaces in Figure 2.1.a. The two pictures represent the same concept. The labels inside the spaces represent tuples. A space can contain other spaces or

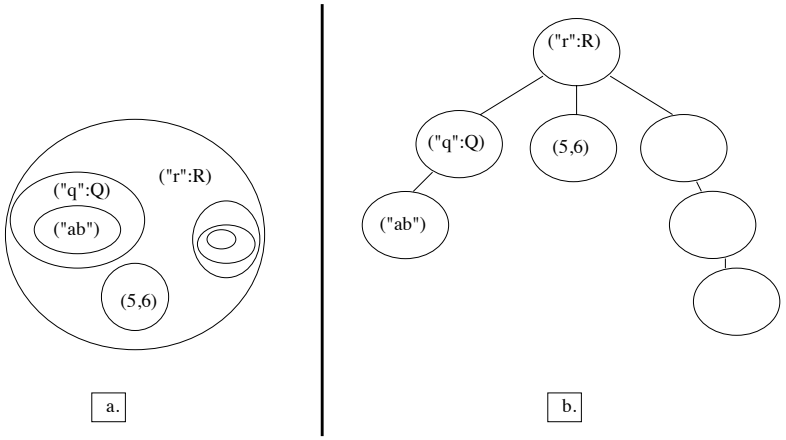

Figure 2.1: PoliS nested spaces (a) and the corresponding tree interpretation (b).

tuples: *ordinary tuples*, which are ordered sequences of values, and *program tuples*, which contain the coordination rules that manage activities inside the space they belong to.

In Figure 2.1 ordered sequences of values (for example $(5, 6)$) are ordinary tuples; the tuples of the form ("r" : R) are program tuples. A program tuple ("r" : R) is composed of an identifier r and rule code represented by the placeholder R. The rule code defines which reactions can take place. The quoted notation " " is used to distinguish actual parameters from formal ones (i.e., the non quoted ones). The execution of a *program tuple* is an action which can modify a space tree by removing and adding tuples. However, an action can only handle the tuples of the space it belongs to *and* the tuples of its parent space. This precisely defines both the "input" and the "output" scope of any action, as represented by a program tuple. Figure 2.2 shows the scope of a program tuple ("r" : R). A space is modified by reactions that transform multi-sets of tuples into multi-sets of tuples (this is multi-set rewriting, common to most coordination models based on *generative communication* [BL96]). A rule defines a reaction that reads and/or consumes tuples in its scope, performs a sequential computation, and produces new tuples in its scope. More precisely, a rule consists of a *precondition*, a *local computation*, and a *postcondition*. The precondition is a multi-set of tuples to be found in the rule scope. The local computation is any sequential computation which does not modify the tuple space; it is encoded as a function that maps values of tuples of the precondition on values of tuples of the postcondition. The postcondition is made up of a multi-set of tuples to be produced in the rule scope. We remark that this is a very general definition; actually a

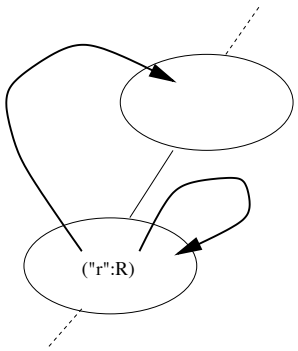

Figure 2.2: The scope of a rule.

rule can lack of some components: a rule can have an empty precondition, can involve no local computation, or can produce no tuples. The precondition can include *formal tuples*, i.e., tuples whose fields can be identifiers (i.e., the non quoted fields). In this case actual values for those identifiers are "matched" in the tuple space.

The tuples of the precondition must be read or consumed in the rule scope (Figure 2.2). When a program tuple is enabled, i.e., its precondition tuples exist in the program tuple scope, the reaction can take place: the tuples to be consumed locally are removed from the space containing the program tuple, the tuples to be consumed externally are removed from the parent space of the space containing the program tuple, the local computation is performed, the tuples of the postcondition are produced. A tuple in the precondition must be *read* if the symbol "?" is put in front of it and must be *consumed* otherwise; a read or consume operation involves the parent space if the symbol "↑" is put in front of a tuple and involves the local space if the symbol is missing; a tuple in the postcondition must be *produced* in the parent space if the symbol " ↑" is put in front of it and must be produced locally otherwise.

Rules are first class entities in PoliS: in fact, they are themselves part of spaces as (program) tuples that can be read, consumed or produced just like ordinary tuples. A program tuple has the form (*"rule_id"*: *rule*) where *rule_id* is a rule identifier and *rule* is PoliS rule code. The identifier simplifies reading or consuming program tuples and allows the existence of multiple copies of program tuples with the same code but different rule identifiers.

Rules can also create and destroy tuple spaces. They can generate new spaces using the primitive **tsc** (for *tuple space creation*) in the postcondition part. For example, the

execution of a rule containing a **tsc**(M) operation in its postcondition causes the space M to be added as a child space of the space where the rule is executed. Spaces can also be destroyed by particular rules called *termination rules*. Whenever a termination rule is enabled the tuple space terminates and disappears. Termination rules can read tuples only locally (i.e., not in the parent space, as the termination condition is meant to be local to the space configuration) and produce tuples in the parent space, as the local space disappears. When the tuples to be read are in the space, the reaction specified by the termination rule takes place in the usual way. Local computation and tuple production are used to communicate possible results to the parent space and then the space terminates. Termination rules are given by means of special program tuples whose names are replaced by the keyword **terminate**. In Figure 2.3.a a new space is created upon the activation of rule R. In Figure 2.3.b a space is destroyed when the termination rule T is enabled.

A simple example helps in explaining both the syntax and the semantics of PoliS. Let us consider a client-server system. A client emits requests and a server serves them. Such a system can be described by two distinct spaces both included in the main space representing the client and the server.

Table 2.1 contains the specification of the system. The *StartContext* space is the main space, that contains the program tuple ("*create*" : *CREATE*). The name of the tuple is *create* (as it is quoted, it is the actual name of the tuple); instead *CREATE* acts as a "macro", expanded in the corresponding text below in the table. The rule denoted by *CREATE* creates the spaces *Client* and *Server* that contain the tuples describing the client and the server, respectively. The rule also consumes the program tuple ("*create*" : *CREATE*) in order to ensure this rule is only applied once in the initialization phase. After that, the code of *CREATE* will disappear and it will not be possible to apply the

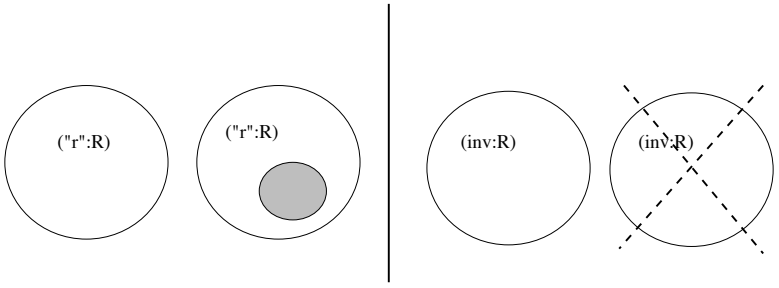

Figure 2.3: Creation (a) and Termination (b) of spaces.

StartContext

$$StartContext = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\}$$

$$CREATE = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\} \longrightarrow \{\!|\ \mathbf{tsc}(Client), \mathbf{tsc}(Server)\ |\!\}$$

Client

$$Client = \{\!|\ (\text{“}idle\text{”}, 0), (\text{“}req\text{”} : REQ), (\text{“}get\text{”} : GET), (\mathbf{terminate} : END)\ |\!\}$$

$$REQ = \{\!|\ (\text{“}idle\text{”}, i)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}request\text{”}, i), (\text{“}wait\text{”}, i)\ |\!\}$$

$$GET = \{\!|\ (\text{“}wait\text{”}, i), \uparrow(\text{“}answer\text{”}, answ, i)\ |\!\} \xrightarrow{(j) \leftarrow f(i)} \{\!|\ (\text{“}idle\text{”}, j)\ |\!\}$$

where $f(x) = (x + 1)$

$$END = \{\!|\ ?(\text{“}idle\text{”}, 10)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}done\text{”})\ |\!\}$$

Server

$$Server = \left\{\!\!\left|\ \begin{array}{l} (\text{“}getreq\text{”} : GETREQ), (\text{“}idle\text{”}), \\ (\text{“}serve\text{”} : SERVE), (\text{“}put\text{”} : PUT) \end{array}\ \right|\!\!\right\}$$

$$GETREQ = \{\!|\ \uparrow(\text{“}request\text{”}, i), (\text{“}idle\text{”})\ |\!\} \longrightarrow \{\!|\ (\text{“}request\text{”}, i)\ |\!\}$$

$$SERVE = \{\!|\ (\text{“}request\text{”}, i)\ |\!\} \longrightarrow \{\!|\ (\text{“}answer\text{”}, answ, i)\ |\!\}$$

$$PUT = \{\!|\ (\text{“}answer\text{”}, answ, i)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}answer\text{”}, answ, i), (\text{“}idle\text{”})\ |\!\}$$

Table 2.1: Specification of a Client-Server System in PoliS.

rule anymore.

Client is the client space and contains the tuple $(\text{“}idle\text{”}, i)$ that indicates the state of the client, the program tuple $(\text{“}req\text{”} : REQ)$, $(\text{“}get\text{”} : GET)$, and the termination rule $(\mathbf{terminate} : END)$ that contains the code of the rules *REQ*, *GET*, and *END* (specified below), respectively. The rule *REQ* emits a new request (tuple) in the main

space: $\uparrow(\text{“}request\text{”}, i)$, and changes the state of the client from $(\text{“}idle\text{”}, i)$ to $(\text{“}wait\text{”}, i)$ where i is the number associated to the request. The rule *GET* waits for an answer in the main space $\uparrow(\text{“}answer\text{”}, answ, i)$ where i corresponds to the number of the request (the rule checks if the tuple $(\text{“}wait\text{”}, i)$ is present). It emits a new state tuple with the number i increased by one by the function f on the arrow in the rule (specified in the **where** clause). The *Client* space terminates as soon as it receives the 10-th answer. The termination rule *END* checks if the *Client* space contains the tuple $(\text{“}idle\text{”}, 10)$, that means that the client has received ten answers from the server. The tuple (“*done*”) represents a termination message sent by the consumer to the main space before dying.

Server is the server space. It contains a tuple denoting its state and three rules: the rule *GETREQ* checks if the state is idle and if a request is present in the main space, then moves the request in the local space. The rule *SERVE* generates an answer to the request. The rule *PUT* resets the state of the server to idle (emitting the tuple (“*idle*”) locally), and move the answer tuple to the main space.

The example above shows that the basic communication mechanisms of PoliS are asynchronous. Rules are transactions and therefore execute in an atomic fashion. They also offer a basic mechanism for synchronization of operations: the rules can atomically read/consume multiple tuples allowing quite complex evolutions. Tuples representing messages are put in the environment by entities which have to communicate. Hence, communication is decoupled because communicating entities do not necessarily know each other; they access tuples by pattern matching. Messages have no destination address, so their contents determine the set of possible receivers. Thus, a space represents at the same time both a component performing computations and a persistent, multicast channel supporting communication among components it contains. Any space communicates with the parent space using a pattern matching mechanism, thus minimizing the assumptions over the the rest of the system.

In the next section we describe the formal operational semantics of PoliS.

2.2 Abstract Syntax and Operational Semantics for PoliS

We describe the semantics of a PoliS specification as the application of simple rewriting operations on multisets. In Table 2.2 we show the abstract syntax for PoliS. A multiset

MS	::=	$\{\!\!\{$ elem $\}\!\!\}$ | MS $\oplus$ MS | MS $\setminus$ MS | (MS)
elem	::=	tuple | MS
tuple	::=	data | program
program	::=	("r" : *Code*)
data	::=	(datalist)
datalist	::=	"*data*" | *value* | "*data*", datalist | *value*, datalist

"r" $\in$ *Ruleid*, the set of rule identifiers

Code $\in$ *Rulecode*, the set of rules code specified in Table 2.4.

In the concrete syntax, *Code* is usually substituted with a macro that expands in the code itself.

value $\in$ *Values*

data $\in$ *String*

Table 2.2: PoliS Abstract Syntax.

(*MS*) is composed of elements that are tuples or multisets, or can be built as the union or difference of multisets. A tuple can be a data tuple or a program tuple. A data tuple is a sequence of values and strings, whereas a program tuple is composed of an identifier and of rule code. The semantics of PoliS is introduced in the Tables 2.3, 2.4, and 2.5. Table 2.3 shows the SOS (Structured Operational Semantics) axioms and rules:

- **L** is a rule describing the local computations. It formalizes the local and isolated evolution of subspaces or subsets of tuples inside a space.

- The axioms **RL**, **RI**, and **T** define the semantics of PoliS rules. These axioms show the reaction taking place when a *program tuple* ("r" : R) is enabled in a space (M). The formalization of the code macro R is shown in Table 2.4 according to the type of R (an example of use of these macro can be found in Table 2.1). R can be a local rule (i.e., R_l), or an interactive rule (i.e., R_i), or a termination rule (i.e., R_{term}). The notation t_c, t_p, t_{ec}, t_{ep} denote the lists of tuples to be consumed locally, produced locally, produced and consumed in the parent space, respectively. $\overline{S}$ is the list of spaces to be created by the rule. $\overline{v}_{\overline{x}}$ and $\overline{v}_{\overline{y}}$ are the formal parameter lists to be substituted by $\overline{x}$ and $\overline{y}$.

  Table 2.3 shows these three types of semantics rules:

– Local rules consume, test, and produce only local tuples, without involving the parent space. The axiom **RL** shows the transition applied on the space M: if the program tuple ("r_l" : R_l) is in M and if the rule R_l is enabled (condition expressed by the predicate *LocEnabled* specified in Table 2.5), the space M is updated deleting the tuples that the rule consumes and adding the tuples (and the new spaces) that the rule produces.

  Table 2.4 contains the specification of R_l.

– Interaction rules interact also with the parent space. The specification of the axiom **RI** shown in Table 2.3 is similar to the one of **RL** just described. Besides updating the space M_1 it updates the space M_2, parent of M_1 as the rule acts on it as well.

– Termination rules, when enabled, cause the termination of the space they are in. These rules have priority over the other rules. Moreover, a termination rule can only test internal tuples and produce external ones; other operations do

L: $\dfrac{MS \longrightarrow MS'}{\{\!|MS|\!\} \longrightarrow \{\!|MS'|\!\}}$

Local Rule

RL: $\{\!|(\text{“}r_l\text{”} : R_l)|\!\} \oplus M \longrightarrow$
$((\{\!|(\text{“}r_l\text{”} : R_l)|\!\} \oplus M) \setminus \{\!|\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}]|\!\}) \oplus \{\!|\overline{t}_p[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}], \overline{S}[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]|\!\}$
if $\mathit{LocEnabled}(R_l, \text{“}r_l\text{”}, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}})$

Interaction Rule

RI: $\{\!|\{\!|(\text{“}r_i\text{”} : R_i)|\!\} \oplus M_1|\!\} \oplus M_2 \longrightarrow$
$\{\!|((\{\!|(\text{“}r_i\text{”} : R_i)|\!\} \oplus M_1) \setminus \{\!|\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}]|\!\}) \oplus \{\!|\overline{t}_p[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}], \overline{S}[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]|\!\}\}$
$\oplus (M_2 \setminus \{\!|\overline{t}_{ec}[\overline{v}_{\overline{x}}/\overline{x}]|\!\}) \oplus \{\!|\overline{t}_{ep}[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]|\!\}$
if $\mathit{IntEnabled}(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}})$

Termination Rule

T: $\{\!|\{\!|(\textbf{terminate} : R_{term})|\!\} \oplus M_1|\!\} \oplus M_2 \longrightarrow M_2 \oplus \{\!|\overline{t}_{ep}[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]|\!\}$
if $\mathit{TermEnabled}(R_{term}, M_1, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \wedge \neg\exists R_t, \overline{v}_{\overline{z}}, \overline{v}_{\overline{k}} | \mathit{TermEnabled}(R_t, M_2, \overline{v}_{\overline{z}}, \overline{v}_{\overline{k}})$

Table 2.3: Structured Operational Semantics Rules and Axioms.

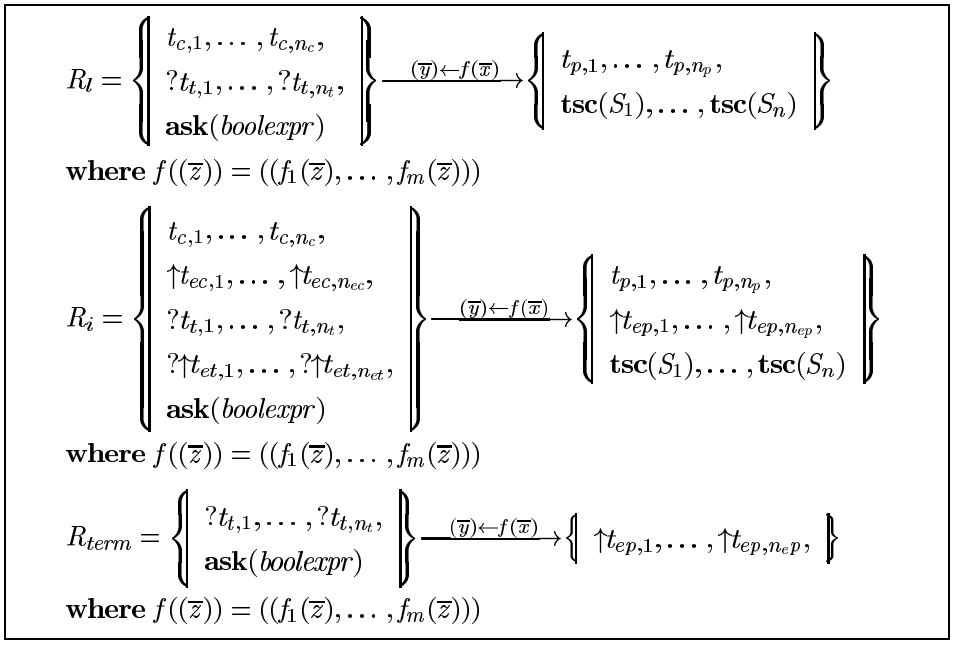

Table 2.4: Classification of PoliS Rules Macro.

not make sense since the local space terminates. The axiom T shows how a space terminates and how some tuples are added to the parent space.

The rule macros R_l, R_i, and R_{term} in Table 2.3 expand as shown in Table 2.4. The notation $(t_{c,1}, \ldots, t_{c,n_c})$ denotes the tuples to be consumed locally, the notation $(?t_{t,1}, \ldots, ?t_{t,n_t})$ denotes the tuples that are tested locally, and $(t_{p,1}, \ldots, t_{p,n_p})$ are the tuples that are produced. $(\textbf{tsc}(S_1), \ldots, \textbf{tsc}(S_n))$ denotes the generated subspaces. In the specification of R_i, and R_{term} the notation $(\uparrow t_{ec,1}, \ldots, \uparrow t_{ec,n_{ec}})$ denotes the tuples consumed in the parent space, while $(? \uparrow t_{et,1}, \ldots, ? \uparrow t_{et,n_{et}})$ are the tuples that are only read from the parent space. Finally, the notation $(\uparrow t_{ep,1}, \ldots, \uparrow t_{ep,n_{ep}})$ denotes the tuples produced in the parent space. The PoliS construct **ask** checks the values of tuples parameters. The general form of the **ask** predicate is **ask**(*predicate*) and it is another condition to be added to the precondition set.

The predicates *TermEnabled*, *LocEnabled*, and *IntEnabled* used in Table 2.3 are formally described in Table 2.5 to check if the rules are enabled. The *TermEnabled* condition is true if the rule R_{term} is enabled, that is, if the program tuple ($term$: R_{term}) is in the same space M as the tuples to be tested. The *LocEnabled* condition

$$
\begin{array}{l}
\mathit{LocEnabled}(R_l, \text{“}r_l\text{”}, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \triangleq \\
\quad \{\!|\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_t[\overline{v}_{\overline{x}}/\overline{x}]|\!\} \subseteq \{\!|(\text{“}r_l\text{”} : R_l)|\!\} \oplus M \\
\quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge \mathit{boolexpr}[\overline{v}_{\overline{x}}/\overline{x}] \\
\quad \wedge\ \forall R, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}} : ((\textbf{terminate} : R) \in M \Rightarrow \\
\quad \neg \mathit{TermEnabled}(R, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \\
\mathit{IntEnabled}(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \triangleq \\
\quad \{\!|\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_t[\overline{v}_{\overline{x}}/\overline{x}]|\!\} \subseteq \{\!|(\text{“}r_i\text{”} : R_i)|\!\} \oplus M_1 \\
\quad \wedge\ \{\!|\overline{t}_{ec}[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_{et}[\overline{v}_{\overline{x}}/\overline{x}]|\!\} \subseteq M_2 \wedge \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge \mathit{boolexpr}[\overline{v}_{\overline{x}}/\overline{x}] \\
\quad \wedge\ \forall R, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}} : ((\textbf{terminate} : R) \in M_1 \Rightarrow \\
\quad \neg \mathit{TermEnabled}(R, M_1, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \\
\quad \wedge\ \forall R, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}} : ((\textbf{terminate} : R) \in M_2 \Rightarrow \\
\quad \neg \mathit{TermEnabled}(R, M_2, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \\
\mathit{TermEnabled}(R_{term}, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \triangleq \\
\quad \{\!|\overline{t}_t[\overline{v}_{\overline{x}}/\overline{x}]|\!\} \subseteq \{\!|(\textbf{terminate} : R_{term})|\!\} \oplus M \\
\quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge \mathit{boolexpr}[\overline{v}_{\overline{x}}/\overline{x}]
\end{array}
$$

Table 2.5: Precondition predicates.

is true if the rule R_l is enabled, that is, if the program tuple (“r” : R_l) is in the same space M as the tuples to be consumed and tested. Furthermore, no termination rules (which have priority) should be enabled. The *IntEnabled* condition is true if the interaction rule R_i is enabled, that is, if the program tuple (“r” : R_i) is in the space M_1 and if the tuple it has to test and consume on the local and parent spaces are respectively in M_1 and in the parent of M_1 (i.e., M_2). Furthermore, no termination rules should be enabled in M_1 or M_2.

For sake of brevity we do not describe the semantics for operator $\oplus$ and $\backslash$; they have their intuitive meanings of multiset union and difference, respectively. We are now ready to define a transition system for PoliS.

$$\mathit{PoliS}_{\mathit{TransitionSystem}} = (MS, \longrightarrow_{MS})$$

where the MS syntax is defined in Table 2.2 and $\longrightarrow_{MS} \subseteq MS \times MS$ is the minimal relation satisfying the rules described above.

The transition system to be associated to a PoliS specification *Spec* is formally defined as a triple
$(\uparrow Spec, \longrightarrow_{Spec}, StartContext)$ where:

- $StartContext$ is the initial MS (called *initial state*)
- $\uparrow Spec \subseteq MS$ is a minimal subset of MS such that:

$$\_\ \frac{MS_1\ \in \uparrow Spec \quad MS_1 \longrightarrow MS_2}{MS_2\ \in \uparrow Spec}$$

- $\longrightarrow_{Spec} \subseteq \uparrow Spec \times \uparrow Spec$ is the restriction of $\longrightarrow$ to $\uparrow Spec$;
- $StartContext\ \in \uparrow Spec$.

The transition system model and the operational semantics have been used for the construction of a model checker for PoliS, that we present in Chapter 3.

2.3 Specification in PoliS

PoliS can be used to specify complex systems. In this section we use it to specify an "Invoicing System". The example was used as a case study in the International Workshop on Comparing System Specification Techniques (Nantes, 1998). two versions of the system are shown. The first does not take into account any interaction with the "environment" while the second considers some possible interactions showing a more complex behavior. An invoicing system should allow the customers to place orders. The orders are then processed by the system, the requested products are provided and together with the invoices sent to the customers. If a product is not available in stock in the ordered quantity, a request is issued and the order is blocked. When the exact quantity becomes available the order is processed. In the next chapter we will use the same example with the model checker and we will prove properties on it.

Startcontext

$$Startcontext = \left\{ \begin{array}{l} Stock, (\text{“}Order\text{”}, 1, p_1, q_1, pending), (\text{“}Order\text{”}, 2, p_2, q_2, pending), \\ ..., (\text{“}Order\text{”}, k, p_k, q_k, pending) \end{array} \right\}$$

Stock

$$Stock = \left\{ \begin{array}{l} (\text{“}Product\text{”}, p_1, q_1), (\text{“}Product\text{”}, p_2, q_2), \\ ..., (\text{“}Product\text{”}, p_n, q_n), (\text{“}invoice\text{”} : IN) \end{array} \right\}$$

$$IN = \left\{ \begin{array}{l} (\text{“}Product\text{”}, p_i, q_i), \\ \textbf{ask}(q_j \leq q_i) \\ \uparrow(\text{“}Order\text{”}, j, p_i, q_j, pending) \end{array} \right\} \xrightarrow{(q_m) \leftarrow f(q_j, q_i)} \left\{ \begin{array}{l} (\text{“}Product\text{”}, p_i, q_m), \\ \uparrow(\text{“}Order, , j, p_i, q_j, invoiced) \end{array} \right\}$$

where $f(q1, q2) = (diff(q1, q2))$

Table 2.6: PoliS specification of Case1

The PoliS specification of the Invoicing System (Case 1)

In order to have a simplified version of the system we suppose to have a closed world: updating, input of new orders, and cancellation of orders have not to be taken into account. Then the initial space will look like the one in Figure 2.4.

The main space contains the orders to be invoiced. Every order is defined by a number (i.e. the *id*), the product reference, the quantity of the product ordered, and the state of the order (*pending/invoiced*).

The *Stock* space is a sub-space of the main space (Figure 2.4). It is specified in the second part of Table 2.3: it contains the stocked products, their names and the stocked quantities. The PoliS rule *IN*, in the *Stock* space, looks for an order in the parent space,

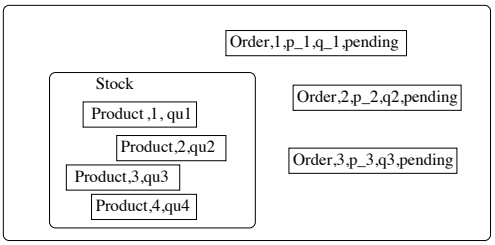

Figure 2.4: Structure of the Invoicing System (Case1).

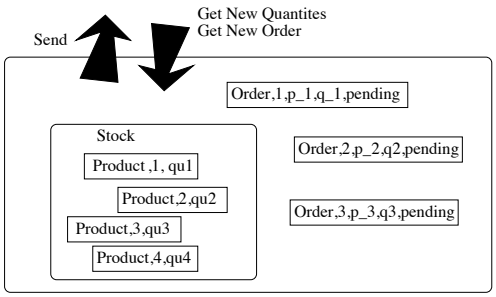

Figure 2.5: Strcuture of the Invoicing System (Case2).

and, if a sufficient quantity of the stocked product exists, it updates the stocked quantity of the product and invoices the order.

Then some questions have risen from this specification. As the rule *IN* checks if the quantity of product stocked is larger than the quantity ordered (by the construct **ask**), the question "What happens if an order asks for a quantity larger than the one present in the stock?" We can ask the customer some details about this situation or proposing a solution: adding a rule that cancel an order in case the asked quantity of product is larger than the stocked one. This implies also the adding of the state *canceled* to the order states *invoice* and *pending*. An other interesting question is "If an order asks for a non-stocked product?". Again, we can choose to handle this case with a rule that cancels the order when it asks for a non-stocked product.

The PoliS specification of the Invoicing System (Case 2)

Now we specify a refinement of Case1, taking into account also the input of new orders, the cancellation of orders and the entries of new quantities in the stock.

In this version of the Invoicing System the main space accepts input from the environment exploiting the operation (⇑ (see Chapter 2 for details on PoliS operators). Figure 2.5 shows the whole system.

Orders can now be in the state *canceled*, besides *pending* and *invoiced*.

The refined specification allows the system to interact with the environment: new orders and new quantities for the stock can be accepted, orders can be canceled (they remain in the space as canceled).

Startcontext

$$Startcontext = \left\{\begin{array}{l} Stock, (\text{“}Order\text{”}, 1, p_1, q_1, pending), (\text{“}Order\text{”}, 2, p_2, q_2, pending), \\ ..., (Order, p_k, k, q_k, pending)(\text{“}ordercounter\text{”}, n), (\text{“}getord\text{”} : GET), \\ (\text{“}canc\text{”} : CANC), (\text{“}newq\text{”} : NEWQUANT), (\text{“}send\text{”} : SEND) \end{array}\right\}$$

$$GET = \left\{\begin{array}{l} (\uparrow(\text{“}neword\text{”}, p, q), \\ (\text{“}ordercounter\text{”}, n) \end{array}\right\} \xrightarrow{(nn) \leftarrow f(n)} \left\{\begin{array}{l} (\text{“}ordercounter\text{”}, nn), \\ (\text{“}Order\text{”}, n, p, q, pending) \end{array}\right\}$$

where $f(n) = (n + 1))$

$$CANC = \left\{\begin{array}{l} \uparrow(\text{“}cancel\text{”}, id_{order}), \\ (\text{“}Order\text{”}, id_{order}, p, q, pending) \end{array}\right\} \longrightarrow \left\{ (\text{“}Order\text{”}, id_{order}, p, q, canceled) \right\}$$

$$NEWQUANT = \left\{ \uparrow(\text{“}newq\text{”}, id_{prod}, q) \right\} \longrightarrow \left\{ (\text{“}newq\text{”}, id_{prod}, q) \right\}$$

$$SEND = \left\{ (\text{“}Order\text{”}, j, p_j, q_j, invoiced) \right\} \longrightarrow \left\{ \uparrow((\text{“}Order\text{”}, j, p_j, q_j, invoiced) \right\}$$

Stock

$$Stock = \left\{\begin{array}{l} (\text{“}Product\text{”}, p_1, q_1), (\text{“}Product\text{”}, p_2, q_2), \\ ..., (\text{“}Product\text{”}, p_n, q_n), (\text{“}invoice\text{”} : IN)(\text{“}update\text{”}, UPDATE) \end{array}\right\}$$

$$IN = \left\{\begin{array}{l} (\text{“}Product\text{”}, p_i, q_i), \\ \textbf{ask}(q_j \leq q_i) \\ \uparrow(\text{“}Order\text{”}, j, p_i, q_j, pending) \end{array}\right\} \xrightarrow{(q_m) \leftarrow f(q_j, q_i)} \left\{\begin{array}{l} (\text{“}Product\text{”}, p_i, q_m), \\ \uparrow(\text{“}Order\text{”}, j, p_i, q_j, invoiced) \end{array}\right\}$$

where $f(q1, q2) = (diff(q1, q2))$

$$UPDATE = \left\{\begin{array}{l} \uparrow(\text{“}newq\text{”}, id_{prod}, q), \\ (\text{“}Product\text{”}, id_{prod}, qi) \end{array}\right\} \xrightarrow{(qnew) \leftarrow f(q, qi)} \left\{ (\text{“}Product\text{”}, id_{prod}, qnew) \right\}$$

where $f(qplus) = (plus(q1, q2))$

Table 2.7: PoliS specification of Case2

The main space now contains new rules for the handling of these situations; the counter tuple introduced records the number of orders accepted (it is updated by the rule *GET*). The rule *GET* accepts a new order recording it as pending. The counter helps in assign sequential identifier to the input orders. The rule *CANC* cancels an order marking it as canceled. The rule *NEWQUANT* simply accept a new quantity of a product as input from

the environment. The rule *SEND* communicate to the environment an invoiced order. The *Stock* space is now enriched with an *UPDATE* rule that updates the product quantities in the stock with the new quantity received from the main space.

Some questions have risen also from this specification. The question risen on the specification of Case1 "What happens if an order asks for a quantity larger than the one present in the stock?" is no more a problem: the input of new quantity of product from the environment allows an order of a larger quantity of product than the stocked one to be invoiced in a future (i.e. when the stocked quantity is updated).

The other question on Case1 ("If an order asks for a non-stocked product?") still is a problem, in fact an order of a non-stocked product will never be invoiced. The specification describes the input of new quantities but not the input of new products. "It is possible to have input of new products?": some rules could be added th handle these situations.

Then a related question is "If a quantity of a non-stocked product arrives?": should this case be considered as "input of new product" or simply "an error of non-stocked product quantity input?". "If the environment asks to cancel a non-present order?" is an other question risen from the specification: the *CANC* rule checks if the order is present and then cancel it. If the order is not present the *CANC* rule is not applied: we could add, if needed, a rule to handle this kind of error signaling it to the environment (as an output). In the next chapter we will analyze this specification using our model checker in order to be able to determine new questions on the specification and to be able to reply to some of them.

2.4 PoliS and Software Architectures

Research in the field of software architecture has led to the definition of several environments and languages for the definition and the design of architecture of software systems. Some works face the problem of defining a general-purpose language for architectural description, supporting system design by correct combination of given interacting subsystems [S+95]. Other works aim to characterize systems design according to defined style constraints, developing style specific environment to guide the building of specific systems [GAO94]. Other architectural description languages have been developed exploiting well-known formalism as CSP [AG97] or π -calculus [MDEK95] providing also tools for

animation and monitoring [L+95].

Software architectures specification is an important phase in the life cycle of software systems. It is universally shared the idea that software architecture specification should be put between the requirement definition and the design phase defining important aspects of systems before actually going into the details of the design itself. In this phase the clear definition of the interaction among different components should be specified.

We now show how PoliS can be used for the specification of software architectures. The basic entity of the PoliS language is the *Tuple-Space*: an architectural component is specified using a space. When necessary, a component can be seen as composition of different sub-components. We specify this kind of compositionality in PoliS exploiting the multiple tuple spaces structure: each composed component is specified with a PoliS space containing other sub-spaces. For instance, a server component can be seen as a single space (as in Table 2.1), or as composed of different entities (i.e. sub-spaces) handling different kind of requests or providing different services: Figure 2.6 shows a server with two handlers for Data-Base queries and WWW services.

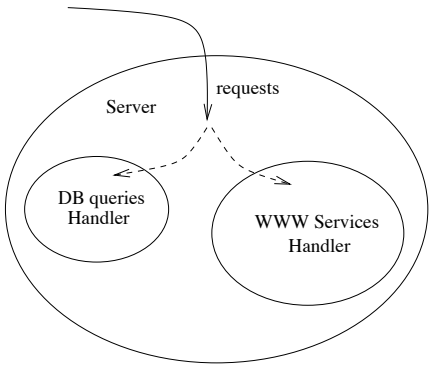

Figure 2.6: Architecture of a Server with two Handlers.

The coordination model is a good framework to abstract from communication details. At the architectural level we would like to have an abstract view of the system: the tuple-based communication mechanism let the focus be put on the structure. On the other side, if the specification of the connection is important, it is possible to associate with the connector a space in order to define its particular behavior. For instance, in the example shown in Table 2.1 the client and the server communicate through the tuple space using this coordination abstraction. We could modify the model adding an entity, with the

function of connector (i.e. a Buffer or a Router) in order to specify its particular behavior. This connector can also be composed of different sub-components, for instance a Layered Router: the nested space model fits the specification of this layered structure (Figure 2.7).

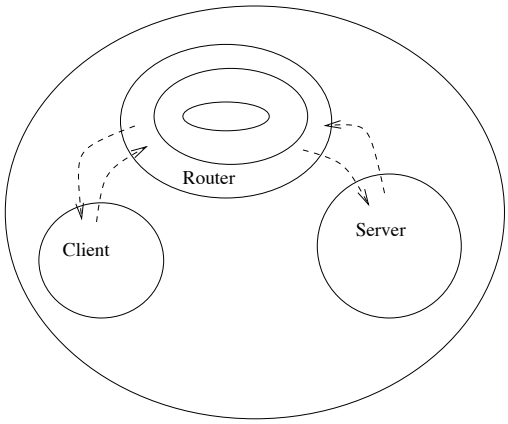

Figure 2.7: Architecture of a Client-Server System with a Layered Router.

The PoliS spaces model allows the specification of context-free components as independent spaces with their active rules. The PoliS mechanism of active rules scoping (see Figure 2.2) helps in the definition of the components assumptions on the external environment. For instance, consider a generic rule enabled only when a particular tuple is present in the parent space ($\uparrow(tuple)$): the component containing that rule should be put in a configuration that will eventually provide that tuple, otherwise parts of the component behavior will be unexploited (with consequences that can lead to the deadlock of the system). In this way we can reason on the assumptions that components make on their contexts and analyze how different assumptions can match and how components can be interconnected. We can predict which context allows a component to behave exploiting all its functions.

These kinds of reasoning could help in the organization of the architectural configuration as also stated in [GKC99]. Furthermore, the help of automatic tools for the testing of these properties could be devised. In this direction we propose the use of our PoliS model checker: we introduce this topic in the next chapter.

Chapter 3

The PoliS Model Checker

In this chapter we introduce the model checker of PoliS that we will use to analyze mobile code systems. We recalls the specifications presented in Chapter 2 to show how systems and software architectures can be analyzed using the model checking approach.

3.1 Model Checking a Coordination Model

Theorem proving has been the most traditional method of system analysis [Bro96]. In theorem proving a deductive system with axioms and derivation rules is usually defined. Starting from the axioms and using the rules it is possible to prove new theorems. Such a method can be applied to software systems as well: if the axiom set is enriched with a formal definition of a software system, then the properties derived from the deductive system are the properties that the system satisfies. In [CMP98] a mapping between the PoliS operational semantics and TLA (Temporal Logic of Action) [Lam94] has been studied. This allowed us to use a theorem prover for formal reasoning on PoliS specifications. However, theorem provers require human interaction in order to complete proofs while model checking techniques provide completely automatic verification frameworks. In this paper we exploit a model checking technique to perform analysis on PoliS specification documents. Model checking was initially used for the verification of hardware systems. A landmark paper [CES86] suggested and studied a model checking approach for software systems. Model checking aims at finding an assignment (*model*) for system variables that satisfies the formulae describing some system properties. Given a model of a software system (derived from its operational specification) a model checker makes an exhaustive

analysis of variable values possible in the model. This method may seem trivial and inefficient, but it is very powerful for systems with finite state models.

Model checkers are completely automatic. An important feature of model checking is the ability to find counter-examples (i.e., a path that leads to a scenario where the property is false). Abstract model checking [CGL94] and deductive model checking [SUM96] are the most often exploited techniques to deal with infinite systems. The exponential explosion of the number of system states can also be managed with symbolic model checking and the use of BDD (Binary Decision Diagrams) [BCM+92].

The model checker we have built exploits PoliS modularity features (i.e., spaces defining context boundaries) in order to reduce the space of the graphs built for a specification. The algorithm applied for the verification of properties follows the one presented in [CES86]. The logic is based on CTL (Computation Tree Logic) [CES86]: the differences between our logic and CTL are related to the spaces-based coordination model. We will give the details of the graph construction, the logic and the model checking in the following sections.

3.1.1 The PoliS Graph Construction

In Section 2.2 we have described an operational semantics for PoliS. We now consider the transition system defined by the Structural Operational Semantics (SOS); the graph obtained from the unfolding of a transition system of a real system is something quite similar to our model. The main difference between SOS unfolding and our model is that in SOS a unique monolithic graph is built to represent a system, while here we associate a graph to each sub-space definition. The nodes of the graph show how a space evolves; instead, edges are labeled with tuples produced/consumed and tested in the parent spaces. As an example consider the space *Component* in Table 3.1 and the graph built for this space in Figure 3.1. The component can be idle or performing some critical actions (when it obtains the token). It can also return the token (by rule *PUT*). The three nodes in Figure 3.1 indicate the possible states for the space *Component*, namely *idle*, *critical*, and *req*. The arrows show the transitions due to the application of the rules *REQ*, *GET*, or *PUT*. The labels on the arrows describe the tuples tested, consumed, and produced in each transition. Our model checker works recursively starting from the more deeply nested spaces, up to the main space.

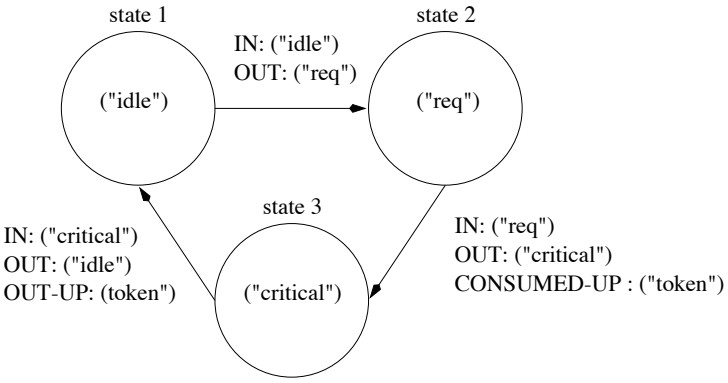

Figure 3.1: Graph for the simple space Component.

We distinguish two kinds of spaces: spaces which do not contain other spaces and spaces which contain subspaces. From hereafter we call *simple spaces* the former, and *compound spaces* the latter. The graph for simple spaces is built according to the SOS transition system. In graphs for compound spaces we exploit *configurations*. A configuration is a triple $(graph, instance, state)$ that uniquely identifies a state in a graph of a space: a configuration is a descriptor for a subspace instance. A graph for a compound space contains a configuration for each subspace. In Table 3.2 we describe the specification for a root space (named *StartContext*) including two instances of space *Component* given in Table 3.1. The graph is built according to the SOS : a labeled transition for each rule activation is built from the initial state (defined by an initial multiset). The final state

Component

$$Component = \left\{\!\!\left\{ \begin{array}{l} (\text{“}idle\text{”}), (\text{“}req\text{”} : REQ), \\ (\text{“}get\text{”} : GET), (\text{“}put\text{”} : PUT) \end{array} \right\}\!\!\right\}$$

$$REQ = \{\!\!\{ (\text{“}idle\text{”}) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}req\text{”}) \}\!\!\}$$

$$GET = \{\!\!\{ (\text{“}req\text{”}), \uparrow(\text{“}token\text{”}) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}critical\text{”}) \}\!\!\}$$

$$PUT = \{\!\!\{ (\text{“}critical\text{”}) \}\!\!\} \longrightarrow \{\!\!\{ (\text{“}idle\text{”}), \uparrow(\text{“}token\text{”}) \}\!\!\}$$

Table 3.1: Specification of a simple component.

StartContext
$StartContext = \{\!\{ \; Component, Component, (\text{“}token\text{”}) \; \}\!\}$

Table 3.2: Specification for *StartContext*.

represents the multiset with rewritten tuples. The transition label includes tuples to be tested, consumed, or produced in the parent space. When a computation is performed inside a subspace, everything in the state representing the parent space is unchanged, but the configuration of the subspace. In Figure 3.2 we show how we exploit configurations[1]: the initial state of the graph corresponding to the main space (*StartContext*) contains the tuple *token* and two configurations corresponding to the two instances of the two components $(C, 1, 1)$, $(C, 2, 1)$, (C stands for *Component*) where the second parameter denotes the instance id (i.e., *Component*1 and *Component*2), and the last parameter is a pointer to the state of the graph of the *Component* space (i.e., the state 1 in both cases).

Our model is more useful and powerful than the SOS model mainly for two reasons: first, we save space when there are several instances of some graph definition, as in the previous example; second, we can abstract a single space and analyze its model independently from other spaces. However, building a graph independently from its context introduces some problems. For example, the case in which a formal tuple has to be consumed in the parent space has to be handled. Uninstantiated identifiers can hold any value, so for correctness, while building the graph, all the cases have to be considered (i.e., all values for each domain). We handled this problem making a guess on a fixed range of natural numbers given with the specification of the system to be analyzed.

In the following we introduce the logic we use to reason on these graphs and then, the details of the model checking tool.

[1]To avoid confusion the transition labels of the figure do not contain the list of the tested, consumed, or produced tuples; instead, we label the edges with the names of the rules applied.

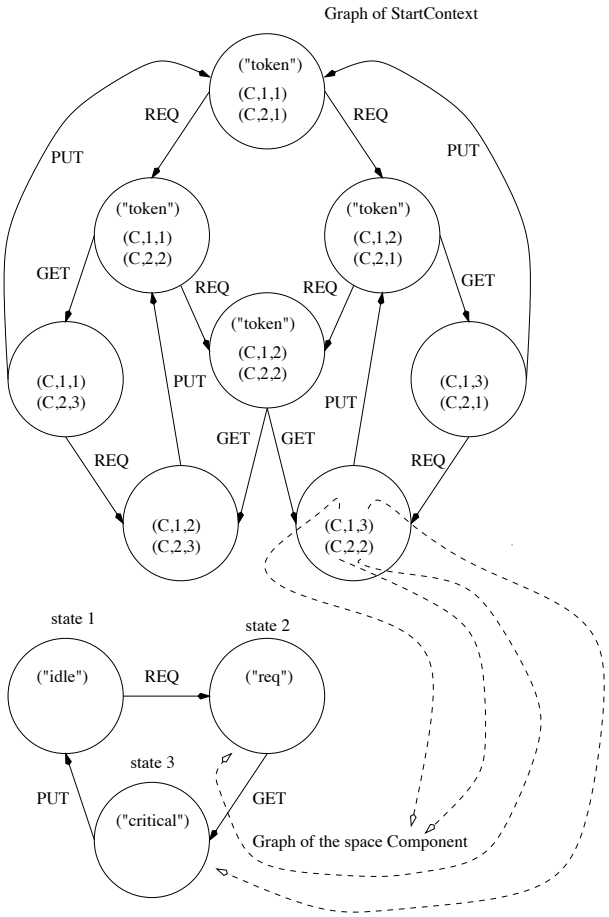

Figure 3.2: Representation of configurations.

3.1.2 The PoliS Temporal Logic

The PoliS Temporal Logic (PTL) is a CTL [CES86] dialect. The main differences between PTL and CTL depend on the definition of our model, that is based on spaces (multi-sets). All the formulae are evaluated in a context (a space); moreover, we assume that formulae without an explicit context are evaluated in the *StartContext*. An atomic proposition *atom* is a tuple. *atom* is true in a context C if the tuple it represents belongs to a space C. We have also added classical logic operators and some temporal operators. In Table 3.3 we sketch the PTL syntax.

- A *ptf* can be a *temporal*, a *classic*, a parenthesized *ptf*, an *atom*, a *ptf* can be universally or existentially quantified over some variables;
- a *context* is a PTL formula that has a pattern like: $ptf \in C$ (space C), $ptf \in \star C$ (all

ptf	::=	context |	temporal	::=	$\star$**X**ptf |
		temporal |			&**X**ptf |
		classic |			$\star$(ptf **U**ptf) |
		(ptf) |			&(ptf **U**ptf) |
		atom |			$\star\Diamond$ptf |
		$\forall$ i $\in$ [min,max] (ptf) |			&$\Diamond$ptf |
		$\exists$ i $\in$ [min,max] (ptf)			$\star\Box$ptf |
context	::=	ptf $\in$ C |			&$\Box$ptf |
		ptf $\in \star$C |			ptf $\rightsquigarrow$ptf
		ptf $\in$ &C |	classic	::=	ptf $\wedge$ptf |
		ptf $\in$ %C			ptf $\vee$ptf |
atom	::=	tuple			$\neg$ptf |
					ptf $\Rightarrow$ptf

Table 3.3: PTL Syntax.

C spaces), *ptf* $\in$ &*C* (at least one C space), or *ptf* $\in$ %*C* (exactly one C space), these because in a specification there can be more than one instance of the same space;

- a *temporal* is a CTL formula: the canonical operators **A** (for all paths) and **E** (at least a path does exist) for path quantification are described respectively by symbols $\star$ and &. **X** and **U** are PTL symbols for CTL operators Next and Until;
- $\star\Diamond$*ptf* is defined as $\star$(true**U***ptf*): it means "*for all paths ptf will be eventually true*";
- &$\Diamond$*ptf* is defined as &(true**U***ptf*): it means "*for at least one path ptf will be eventually true*";
- $\star\Box$ptf is defined as $\neg$&$\Diamond\neg$ *ptf* : it means that "*for all paths ptf is always true*";
- &$\Box$*ptf* is defined as $\neg\star\Diamond\neg$ *ptf* : it means that "*for at least a path ptf is always true*";
- *ptf* $\rightsquigarrow$*ptf'* is defined as $\star\Box$(*ptf* $\Rightarrow$ $\star\Diamond$*ptf'*): it means that "*for all paths it is always true that ptf implies that for all paths ptf' will be eventually true*";
- a *classic* is a PTL formula with classical logic operators;
- an *atom* is simply a tuple.

3.1.3 The PoliS Model Checker

We now describe the details of our model checking tool. PoliMC is our model checker for PoliS. The model checker gets two inputs: a system specification written in PoliS, and a set of properties to be verified, written in PTL. PoliMC first parses the PoliS specification and builds up a model for it as described in Section 3.1.1; then, it parses the PTL formulae and builds syntactic trees. Finally, it starts the model checking phase.

The model checking algorithm we apply follows the guidelines given by Clarke in [CES86]. As we have shown in Section 3.1.2 all formulae can be rewritten using these operators: **X**,**U**(preceded by & or $\star$),$\wedge$,$\neg$. Thus, the only temporal formulae to verify are of the form:

$$p \wedge\ q,\ \neg\ p,\ \&\mathbf{X}p,\ \star\mathbf{X}p,\ \&(p\mathbf{U}q),\ \star(p\mathbf{U}q)$$

The quantified formulae are handled like macros. A universally quantified formula is expanded in a logic conjunction of its sub-formulae while an existentially quantified formula is substituted by the logic disjunction of its sub-formulae. For instance:

$$\forall\ i \in [0,3]\ (ptf_i)\ \equiv\ (ptf_0 \wedge ptf_1 \wedge ptf_2 \wedge ptf_3)$$

$$\exists\ i \in [0,3]\ (ptf_i)\ \equiv\ (ptf_0 \vee ptf_1 \vee ptf_2 \vee ptf_3)$$

The main difference with respect to the Clarke algorithm is in the handling of *context* formulae. Each sub-formula is checked inside its context. When, during the checking, PoliMC finds a *context* formula like $p \in C$, it leaves the current graph and it starts checking the graph bound to C. This task is performed recursively. When the checking is finished the currently checked state of the parent graph which contains a configuration $(state, graph, instance)$ where *graph* is bound to C and *state* is a state of the graph satisfying p, is labeled with the formula $p \in C$.

The verification of a formula of the form $p \in \{\%, \&, \star\}C$ is similar: a state of the parent of the graph bound to C can be labeled if among all the configurations it contains, which have the *graph* component bound to C, there is respectively only one configuration, some configurations, or all the configurations satisfying the formula p.

The verification of a formula is performed bottom-up: if the length of the formula is n, PoliMC first checks all the sub-formulae of length less than n, then it labels each state according to the labeling of the sub-formulae.

The verification of *atom* formulae is trivial: an *atom* is a tuple and a state will be labeled with this formula if and only if it represents a space which contains the tuple.

The verification of formulae $p \wedge q$, and $\neg q$ depends on the verification of p and q. $\&\mathbf{X}\ p$ and $\star\mathbf{X}\ p$ can be easily checked too: a state s is labeled with this formulae if some or all the states s' in the transitions of type (s, s') are labeled with p.

The verification of formulae that contain the until ($\mathbf{U}$) operator is more complex. The check for $\star(p\ \mathbf{U}\ q)$ is done forward, while the check for $\&(p\ \mathbf{U}\ q)$ is done backward, operating recursively. According to the $\mathbf{U}$ definition a state s can be labeled with $\star(p\ \mathbf{U}\ q)$ if s is labeled with q, or if it is labeled with p and all its successor states are labeled with $\star(p\ \mathbf{U}\ q)$. On the contrary, a state s can be labeled with $\&(p\ \mathbf{U}\ q)$ if it is labeled with q or if it is labeled with p and one of its successor states is labeled with $\&(p\ \mathbf{U}\ q)$. We remark that if a specification contains the creation of new spaces we could obtain infinite graphs, thus we use model checking on a constrained version of the specification where we limit the number of possible generated spaces. This implies that we cannot "prove" properties on the model, as we do not explore all the possible paths. Therefore, we use the tool to test the specifications, to see if formulas are satisfiable, and to find counter-examples.

3.2 Analysis of the Invoice System

The PoliS specification language allows the completely separated specification of the coordination aspects of a system from the computational ones: the specification of functional aspects is limited to functions f in the rules. For instance, the rule *IN* (Table 2.3) specifies the operation of invoicing an order defining the modification of the tuple *Order*, and the functional operation of calculating the difference between the quantity q of the product in the stock and the required quantity $q1$ (indicated on the top of the arrow of the rule).

The multiple spaces based model encodes in a modular way the different components of the system (e.g. the *Stock* is a space). If the requirements had said it, a particular space containing all the orders (i.e. an abstraction of a commercial office) would have been defined.

The input from the environment is seen in PoliS as inheritance of tuples by the main space from an hypothetic external space. The parallel handling of tuples is suitable for this study: it simulates the parallel handling of the orders (or of the updating of the product

quantities) by different employees in a company. It is possible to modify the specification creating different agents (the employees) that handle orders (or updates the quantities). The agents can be modeled in PoliS as spaces where the operations of invoicing orders can take place. In this way the process of invoicing in a company can be specified taking into account the personnel availability and the different roles. PoliS allows also the splitting of the stock in different sub-stocks containing different kind of products (one single product per stock or similar products in the same stock): multiple spaces would be added and rules would help in the search of the right stock for an order.

The parser which is part of the model checking tool helps in finding syntax errors and wrong constructs or sentences. After having checked the specification against these errors the model checker tries to build the graph of the possible evolutions of the system. Obviously, while using a model checker, you have to limit the scope of the variables to a finite set of values in order to generate finite graphs [CES86].

We have tried to verify some liveness properties on the two versions of the Invoicing System. The model checker helped us in detecting some errors in our reasoning. The first error we found was in the specification. The model checker revealed some difficulties in building the graph representing the system evolutions: it tried to generate negative numbers for the quantities q of the products. The model checker helped us in debugging the specification and we found out that we did not put the constrain $\mathbf{ask}(q_j \leq q_i)$ on the specification of the rule *IN* (invoice) (Table 2.3). This condition checks that the quantity of product in the stock is larger than the ordered one: only under this condition an order can be invoiced. After having successfully built the graph of the two specifications we tried to verify the property (3.1) on the first system:

$$\forall p, q, id(\text{“}Order\text{”}, id, p, q, pending) \in StartContext \rightsquigarrow (\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext \tag{3.1}$$

That is, if an order has to be invoiced, it will eventually be invoiced ($\rightsquigarrow$ stands for “leads to”). However, the model checker verified that property (3.1) is false. In fact we have to ensure that the product requested (p) is present in the stock at least in quantity

q:

$$\begin{aligned}\forall id, p, q, \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \wedge \\ (\text{“}Product\text{”}, p, q + k) \in Stock) \rightsquigarrow \\ (\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext\end{aligned} \tag{3.2}$$

By the way, that condition is not enough yet: the model checker still found out that it is false. The reason is that there can be other orders for the same product that can be invoiced before the order id, and that do not leave in the stock enough quantity of product p to let the order id be invoiced. Then, we rewrote the (3.2) as:

$$\begin{aligned}\forall id, id', p, q, q' \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \\ \wedge(\text{“}Product\text{”}, p, q + k) \in Stock \wedge id \neq id' \wedge \\ (\neg(\text{“}Order\text{”}, id', p, q', pending) \in StartContext)) \rightsquigarrow \\ (\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext\end{aligned} \tag{3.3}$$

Then, the model checker verified (3.3). We then tried to verify a property on the second system (Table 2.7). In this system input from the environment is accepted, so we had to take into account the updating of the products stocked (see Table 2.7: rule $NEWQUANT$):

$$\begin{aligned}\forall id, id', p, q, q' \exists k((\text{“}Order\text{”}, id, p, q, pending) \in StartContext \wedge \\ (\text{“}Product\text{”}, p, q + k) \in Stock \wedge id \neq id' \wedge \\ \star(\neg((\text{“}Order\text{”}, id', p, q', pending) \in StartContext) \mathbf{U} \\ \neg((\text{“}Order\text{”}, id, p, q, pending) \in StartContext)) \\ \rightsquigarrow((\text{“}Order\text{”}, id, p, q, invoiced) \in StartContext \\ \vee(\text{“}Order\text{”}, id, p, q, canceled) \in StartContext)\end{aligned} \tag{3.4}$$

That is, if an order of product p is to be invoiced, and there is enough quantity of product in the stock, and a new order for the same product is not accepted until ($\mathbf{U}$ stands for until) the order id is to be invoiced, then, eventually the order will be invoiced or canceled.

In a previous proof session we tried to verify that, at these conditions, the order will eventually be invoiced. However the model checker verified that this property was false: we had to take into account also order cancellations.

PoliS offers a different approach with respect to languages like Z [Spi92] or VDM. These languages are property oriented and have a declarative approach. Z is a very expressive notation and strongly typed: type checking helps in dealing with large specification documents where type errors are more frequent. PoliS is a type-less language, the parser and the model checker detect syntax errors and verify temporal properties, however no type checking can be performed on the specification. On the other hand Z language hardly specifies dynamics of a system: many enhancements and integrations with other notations have been tried in order to allow dynamic aspects to be specified [Eva94, CCM97]. PoliS emphasizes the behavioral aspects of a system, highlighting the rules configuration.

The PoliS operational model helped in understanding the dynamics of the Invoicing System: Case 1 consists of a simple specification containing a single rule (IN) that helps in invoicing the orders decreasing the quantities of products in the stock. In Case 2 the environment has a role and the input of new orders and new quantities of products are considered. The specification is more complex than in Case 1: there are new rules that handle the input and the output with the environment.

Some questions have risen from both the specifications. from the Case 1 specification: "What happens if an order asks for a non-stocked product? Or for a quantity of product larger than the one stocked?". From Case 2: "What happens if a quantity of non-stocked product arrives? Or if the environment asks to cancel a non-present order?". The model checking technique helped us in inferring some properties on our specifications and in increasing confidence in the dynamics: trying to proving properties, helps in finding comprehension errors and features of the model.

3.3 Model Checking Software Architectures

In this section we outline a technique to check the behavior of components as isolated from the context. We can also make interesting proofs on the properties of composed architectures, where the components analyzed before are put in relation and interact. The configuration matching can be performed on multiple components.

This sort of analysis is possible as the model checker works bottom-up on the spaces, building graphs for the innermost ones and then going on recursively. Other key issues in this sort of compositionality analysis are the assumptions that a space (i.e. component) makes on the environment. The PoliS language provides a particular scoping mechanism: the reactions contained in a space can make assumptions on the external space (i.e. the parent of the local space) using the $\uparrow$ operator and formal tuples (not instanced) (see Chapter 2 for details).

A component (i.e a space) that is put in a context (i.e. an other space) uses pattern matching mechanism to match the assumptions contained in its rules (i.e. the tuples with "$\uparrow$") with the actual tuples contained in the environment. In this way we can easily state when a component will be able to have an useful behavior exploiting its functionalities and when not. If the environment does not provide the tuples that the component needs, the behavior of the component will be constrained and its capabilities will not be completely exploited. In a previous work [CMP98] a mapping between PoliS operational semantics and TLA (Temporal Logic of Action) has been studied. This allowed us to use a theorem prover for formal reasoning on PoliS specifications. In this work instead we exploit a model checking technique to perform architectural analysis on PoliS specification documents.

We show how the model checker can be used for the verification of properties on software architectures. We first analyze single components out of their context, considering their interactions with the environment. Then we will be able to analyze configurations and saying if they are feasible and convenient. The study of components as isolated entities is useful when dealing with complex architectures where components are not elementary objects but they are composed of many parts.

We now show how a single component can be analyzed out of its context. Consider the Server in the Client-Server example (2.1): the Server makes only one assumption on the external context, that is, it remains idle until a request is present in its context (i.e. the father space) ($\uparrow$("*request*", i)), then a *GETREQ* reaction can take place and after some steps an answer is generated in the environment ($\uparrow$(*answer*, *answ*, i)). The model checker can be used to prove this property:

$$\forall i, a, C((\text{“}request\text{”}, i), Server) \in C \rightsquigarrow (\text{“}answer\text{”}, a, i) \in C \tag{3.5}$$

That is, if the context C of the Server guarantees the arrival of a request, then the answer to the request will be provided. The Client can emit a request without checking the context C, however it blocks if the context does not provide an answer (rule GET). Then, if an answer is provided the Client can go on making requests till the number of requested services is ten.

$$\forall i, a, C(Client, (\text{“}answer\text{”}, a, i)) \in C \rightsquigarrow \\ (((\text{“}request\text{”}, i+1) \in C) \vee (\text{“}done\text{”}) \in C) \tag{3.6}$$

We can put together the assumptions of the two components and try to check if our Client-Server configuration is feasible.

$$(Client, Server, (\text{“}request\text{”}, i) \wedge (i < 10)) \in C \rightsquigarrow \tag{3.7}$$

$$(\text{“}answer\text{”}, a, i) \in C \rightsquigarrow \tag{3.8}$$

$$((\text{“}request\text{”}, i+1) \in C \vee (\text{“}done\text{”}) \in C) \tag{3.9}$$

We can trivially reach a state satisfying (3.7) in fact the Client can emit a request (with $i < 10$). The first “leads to” ($\rightsquigarrow$) property is satisfied by (3.5) as just shown, and the second “leads to” property is satisfied by (3.6). Hence, we conclude that the two components form a feasible configuration and that the corresponding assumptions match.

The Client-Server is a simple example without reconfiguration problems due to mobility of components. The introduced approach of analysis can be very useful to know if a mobile component could be introduced or not in a particular sub-architecture. For instance, if we introduce an agent in our Client site (space) and want to send it to the Server site in order to avoid heavy communication due to exchanging of requests-replies messages, we could analyze the Agent space and its assumptions on the environment and see if they match with the Server space contents.

In the next chapter we show how systems containing mobile components can be specified and analyzed using PoliS and the model checker.

Chapter 4

PoliS Specification and Analysis of Mobile Code Systems

Modern network technologies including mobile computers and devices, and the programming languages for the Internet, like Java, require novel software design techniques. An important feature in network applications is mobility; however, it is still unclear which entities can be mobile and especially why and when they should move over the network. Mobility can range from mobility of data, as in client-server architectures, to mobility of code, as in Java based applications, to mobility of agents, as in some applications for electronic commerce, to mobility of whole operating environments, as in platforms including mobile hardware.

In this section we show how PoliS can be used for the specification of systems containing mobile components, and in the next sections we illustrate how we use our model checker to analyze mobile systems.

The PoliS language allows the specification of both data and code mobility as first class operations. Mobility of data is denoted by rules able to consume tuples locally and to produce tuples outside the local space (or vice versa). Code mobility is denoted by rules able to consume and produce tuples containing code, i.e., other rules. The ability of moving code and data and the creation/destruction operations acting on spaces allow the specification of mobility of complex agents carrying code and data as well.

Agents in this context are represented by spaces containing tuples and rules. Agent mobility is coded by a combination of code and data mobility. In order to show how agent mobility can be expressed in PoliS we modify the example in Table 2.1 by adding an agent that is sent from the client to the server to perform some computations (Figure 4.1). The

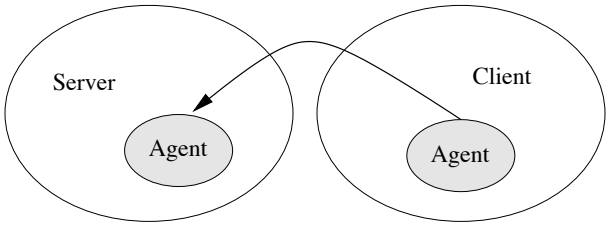

Figure 4.1: A simple Client-Server system with a Mobile Agent.

example shows how we specify mobility of data, code, and agents. Tables 4.1 and 4.2 contains the PoliS specification of the system.

StartContext

$StartContext = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\}$

$CREATE = \{\!|\ (\text{“}create\text{”} : CREATE)\ |\!\} \longrightarrow \{\!|\ \mathbf{tsc}(Client), \mathbf{tsc}(Server)\ |\!\}$

Client

$Client = \{\!|\ Agent, (\text{“}get\_ready\text{”}), (\text{“}send\text{”} : SEND)\ |\!\}$

$SEND = \{\!|\ (\text{“}frozen\text{”}, t), (\text{“}agent\text{”} : a)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}agent\text{”} : a), \uparrow(\text{“}frozen\text{”}, t), (\text{“}wait\text{”})\ |\!\}$

Server

$Server = \{\!|\ (\text{“}getag\text{”} : GET), (\text{“}data\text{”}, \text{“}d\text{”})\ |\!\}$

$GET = \{\!|\ \uparrow(\text{“}frozen\text{”}, t), \uparrow(agent : a)\ |\!\} \longrightarrow \{\!|\ (\text{“}frozen\text{”}, t), (agent : a), (\text{“}unfreeze\text{”})\ |\!\}$

Table 4.1: Specification of a Client-Server System with a Mobile Agent: first part.

Client is the client space. It contains the subspace *Agent*, the tuple (“*get_ready*”), and

Agent

$$Agent = \left\{ \begin{array}{l} (\text{“}start\text{”} : START), (\text{“}resume\text{”} : RESUME), (\text{“}job\text{”} : JOB), \\ (\textbf{terminate} : READY), (\text{“}agent\text{”} : AGENT) \end{array} \right\}$$

$$START = \left\{ \uparrow(\text{“}get\_ready\text{”}) \right\} \longrightarrow \left\{ (\text{“}state\text{”}, \text{“}readytogo\text{”}, 0) \right\}$$

$$READY = \left\{ \begin{array}{l} ?(\text{“}state\text{”}, s, r), ?(\text{“}agent\text{”} : AGENT), \\ \textbf{ask}(s = \text{“}readytogo\text{”} \ or \ s = \text{“}afterjob\text{”}) \end{array} \right\} \xrightarrow{(t) \leftarrow f(s,r)} \left\{ \begin{array}{l} \uparrow(\text{“}frozen\text{”}, t), \\ \uparrow(\text{“}agent\text{”} : AGENT) \end{array} \right\}$$

where $f(state, result) = (freeze(state, result))$

$$RESUME = \left\{ \uparrow(\text{“}frozen\text{”}, t) \right\} \xrightarrow{(s,r) \leftarrow f(t)} \left\{ (\text{“}state\text{”}, s, r) \right\}$$

where $f(tuple) = (\text{“}beforejob\text{”}, result(tuple))$

$$JOB = \left\{ \begin{array}{l} (\text{“}state\text{”}, s, r), \uparrow(\text{“}data\text{”}, d), \\ \textbf{ask}(s = \text{“}beforejob\text{”}) \end{array} \right\} \xrightarrow{(r',s') \leftarrow f(d,r,s)} \left\{ (\text{“}state\text{”}, s', r') \right\}$$

where $f(data, oldresult, \text{“}beforejob\text{”}) = (calculate(data, oldresult), \text{“}afterjob\text{”}))$

$$AGENT = \left\{ (\text{“}unfreeze\text{”}), (\text{“}agent\text{”} : AGENT) \right\} \longrightarrow \left\{ \textbf{tsc}(Agent) \right\}$$

Table 4.2: Specification of a Client-Server System with a Mobile Agent: second part.

the program tuple (“*send*” : *SEND*). The data tuple (“*get_ready*”) tells the agent to get ready to be sent. The code of the rule *SEND* actually sends the agent (once ready), i.e., the tuple *frozen* and a program tuple (“*agent*” : a), where a is a formal parameter that is matched with a piece of code (in this case the code *AGENT* when present).

The *Agent* space is described in the same table. The rule *START* consumes the data tuple (“*get_ready*”) from the client space (i.e., the parent space) and produces the tuple (“*state*”, “*readytogo*”, 0) into the agent local space enabling the rule *READY* for execution. The termination rule *READY* terminates the space saving the status of the agent, i.e., the *frozen* tuple, and the activation rule (“*agent*” : *AGENT*) in the client space. The predicate **ask** checks if the value of s (contained in the *state* tuple) is either *readytogo* or *afterjob*, i.e., the agent is ready to be sent, or it has finished a job. This enables the client’s rule *SEND*, already described. The *Agent* space also contains the program tuples (“*agent*” : *AGENT*), (“*resume*” : *RESUME*), and (“*job*” : *JOB*). The first acts as an “unfreeze” for the agent space whenever the agent is “frozen” (i.e. it generates the new space). The rule *RESUME* reacts when the agent space has been created, getting the frozen status of the agent

and emitting the status tuple in the agent space. The rule *JOB* denotes the real code for executing a job: it is used when the agent is at the server site and the "data" are available. The **where** clause is abstractly specified as a function f because in the spirit of most coordination languages (which separate computation from coordination) we omit computation details, however it could be refined defining the exact mechanism for the computation of the result. The *Server* space contains the data that will be used by the agent code *JOB* to compute a result, and the program tuple ("*getag*" : *GET*), to gather an agent from the environment. The rule *GET* gathers the frozen agent and the activation code (i.e.,("*agent*" : a), with the formal parameter a matching the real code), and emits the tuple ("*unfreeze*") so that the agent can unfreeze itself.

As the example shows, code mobility can be modeled in PoliS consuming and producing tuples representing code (i.e., containing rules). For instance, the rule *SEND* consumes locally and produces in the environment the program tuple containing the activation code for the agent, i.e., ("*agent*" : a), where a is a formal parameter matched with the code *AGENT* when available. Agent mobility is depicted in Figure 4.2. An agent is "frozen" and the code for the re-activation of the agent is moved together with its frozen status to another location, where the agent will be reactivated.

This approach to agent mobility has several advantages. PoliS shows clearly that code and state mobility are orthogonal concepts. For instance, we can specify the movement of several agents sharing the same code simply using as many status tuples as agents and a single code tuple. Another example is that we can redefine the behavior of an agent changing its code but keeping its state. Another advantage is related to the performance of the model checker we have implemented for the language (see Section 3.1): the consideration of space (i.e., agent) mobility as first class in the language on one hand would allow rules to consume and produce spaces as normal tuples. On the other this would lead to an explosion in the number of states to be considered by the tool. Nevertheless, we are exploring the possibility of enhancing the language with space mobility and studying how we can still reason automatically on such a model.

The basic mobility mechanism we have in PoliS is constrained to be "step by step", that is no general visibility on all the possible locations is considered. Agents can be either "pushed" to known locations, or "pulled" inside a space by the space itself. A tuple of data or code can be moved from one space to the parent, or pulled from the parent

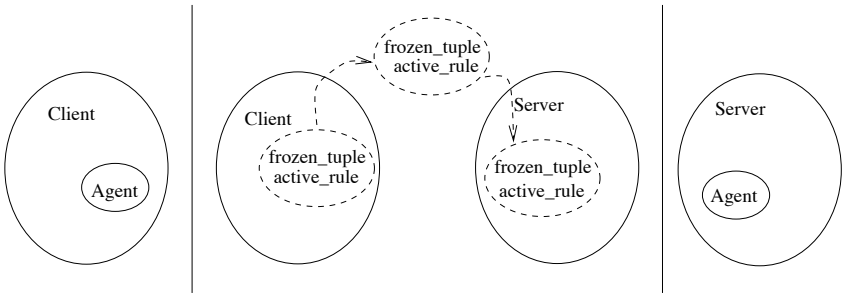

Figure 4.2: Agent Mobility in PoliS.

to the local space and a complex path composed of these steps can be generated. This abstraction models a general network architecture including layered routers, hierarchical LANs structure, and fire-walls [CG00]. The ability to dynamic re-arrange the hierarchy of spaces allows a strong control on agent interactions. Moreover, from a model checking perspective the "step by step" mobility mechanism permits a more constrained space explosion than with a general "move to location" mechanism.

4.1 Specification of an Architecture with Mobile Agents

We use PoliS to specify a "Meeting Scheduler System" including mobile agents. This problem was proposed as a case study in mobility for the International Workshop on Software Specification and Design [FFFv97]. We first give an informal description (Sect. 4.1.1), then a PoliS specification (Sect. 4.1.2).

4.1.1 The Meeting Scheduler System: an Informal Description

An organization manages meetings as follows. A meeting initiator asks all potential attendees for the following information to be included in their personal agendas:

- a set of dates on which they cannot attend a meeting (exclusion set);
- a set of dates on which they would prefer a meeting to take place (preference set). For simplicity, and without loss of generality, we assume that all days outside the exclusion set and not yet fixed for a meeting are free and represent the preference set.

The proposed meeting date should belong to none of the exclusion sets and to as many preference sets as possible. A date conflict occurs when no date can be found. Conflicts can be resolved in two ways:

- some participants remove some dates from their exclusion set;
- some participants withdraw from the meeting.

The system should assist users in the following activities.

- Plan meetings consistently, using the constraints expressed by participants.
- Re-plan a meeting dynamically (to offer flexibility). Participants should be allowed to modify their exclusion and preference sets before a meeting date is decided. A meeting date initially found may need to be modified; sometimes the meeting may even be canceled.
- Support conflict resolution according to some arbitrary resolution policies.

The meeting scheduler system must in general handle several meeting requests in parallel. Meeting requests can compete by overlapping in time: concurrency must thus be managed.

Admittedly, this problem can be solved with more conventional technologies: there is no need of mobile agents if we centralize all data in some "meeting server". The main advantage of using mobile agents is that an agent can exploit reliable links to travel and perform local computations on a site avoiding movement when, for instance, the net is congestioned. We use this case study only to show how PoliS can be used to deal with a solution based on mobile agents.

4.1.2 A Specification including Mobile Agents

The "Meeting Scheduler System" specification document in PoliS is organized as follows: every initiator of a meeting is associated to a multi-set of tuples representing a mobile agent. Several agents (one for each meeting) can run in parallel. Each initiator agent moves among the sites of participants collecting preferences and trying to decide a date (see Fig. 4.3). For simplicity we assume that a meeting can take place only if all potential attendees will participate. An agent collects information inside a participant space, then it is frozen and moved outside the space:

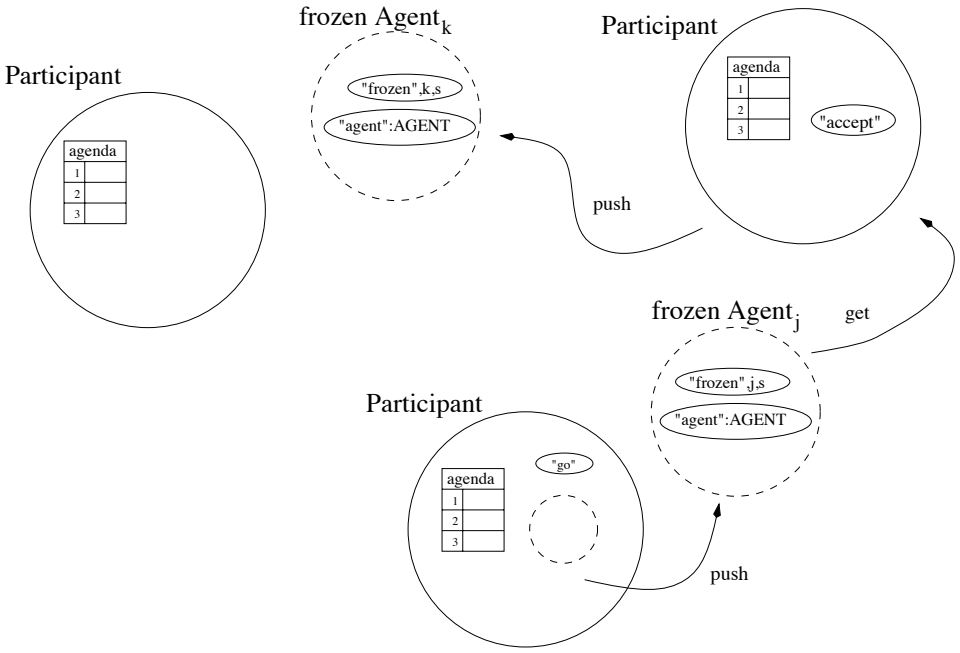

Figure 4.3: Agents System Architecture.

Tables 4.3, 4.4, and 4.5 show the specification of three kinds of spaces. The *StartContext* (Table 4.3) is the initial space: it includes p participants and n agents, one for each meeting. Each *Participant* space (Table 4.4) has an initial state consisting of tuples representing its agenda: some days are marked "free" and others are marked "exclusion", meaning that these dates are in the participant exclusion set (we implicitly assume that the number of meetings (n) is less or equal to the number of days in the agenda (m)). Agendas are represented by the multi-sets after the $\oplus$ operator in the *StartContext* definition of Table 4.3. Tuples ("*start*", n) are consumed by agents to prepare themselves for the shipping, get a self identifier n and start migrating (see rule *START* in the *Agent* space).

The *StartContext* space includes just the program tuple ("*end*" : *END*): the code of the rule *END* associated to the program tuple checks that all the potential attendees will participate, that is the condition for the meeting to take place (the function f_{end} checks if the number of participants has reached a given number and outputs a date).

Each *Participant* space can accept incoming *agents*. It contains some program tuples to activate the following rules. The rule *GETAG* allows the agent to enter in a space. It consumes the tuples ("*frozen*", h, s) and ("*agent*" : a) from the main space and generates them locally. It also consumes the ("*accept*") tuple locally and generates

StartContext

$$StartContext = \left\{\!\!\left\{ \begin{array}{l} Participant \oplus \{\!\{(\text{“}day\text{”}, 1, \text{“}free\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}exclusion\text{”})\}\!\}, \\ Participant \oplus \{\!\{\ldots\}\!\}, \ldots \\ Participant \oplus \{\!\{(\text{“}day\text{”}, 1, \text{“}exclusion\text{”}), \ldots, (\text{“}day\text{”}, m, \text{“}free\text{”})\}\!\}, \\ Agent, \ldots, Agent, \\ (\text{“}start\text{”}, 1), \ldots, (\text{“}start\text{”}, n), \\ (\text{“}end\text{”} : END) \end{array} \right\}\!\!\right\}$$

$$END = \{\!\{\ (\text{“}frozen\text{”}, k, s)\ \}\!\} \xrightarrow{(day) \leftarrow f(s)} \{\!\{\ (\text{“}end\text{”}, day, k)\ \}\!\}$$

where $f(x) = (f_{end}(x))$

Table 4.3: The Meeting Scheduler: the Main Space.

Participant

$$Participant = \left\{\!\!\left\{ \begin{array}{l} (\text{“}get\text{”} : GETAG), (\text{“}push\text{”} : PUSHAG), \\ (\text{“}extend\text{”} : EXTEND), (\text{“}accept\text{”}) \end{array} \right\}\!\!\right\}$$

$$GETAG = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}frozen\text{”}, h, s), \\ \uparrow(\text{“}agent\text{”} : a), (\text{“}accept\text{”}) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \begin{array}{l} (\text{“}frozen\text{”}, h, s), \\ (\text{“}agent\text{”} : a), (\text{“}agent\text{”}) \end{array} \right\}\!\!\right\}$$

$$PUSHAG = \left\{\!\!\left\{ \begin{array}{l} (\text{“}frozen\text{”}, h, s), \\ (\text{“}agent\text{”} : a), (\text{“}go\text{”}) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}frozen\text{”}, h, s), \\ \uparrow(\text{“}agent\text{”} : a), (\text{“}accept\text{”}) \end{array} \right\}\!\!\right\}$$

$$EXTEND = \left\{\!\!\left\{ \begin{array}{l} (\text{“}day\text{”}, d, \text{“}exclusion\text{”}), \\ ?(\text{“}accept\text{”}) \end{array} \right\}\!\!\right\} \longrightarrow \{\!\{\ (\text{“}day\text{”}, d, \text{“}free\text{”})\ \}\!\}$$

Table 4.4: The Meeting Scheduler: the Participant Space.

the tuple (“*agent*”), meaning that the frozen agent has been entered in the local space. The rule *PUSHAG* moves the agent out of a space. It moves the tuples (“*agent*”) and (“*frozen*”, h, s) to the main space. Fig. 4.3 shows the actions of the two rules. Participants can extend the set of possible dates using the rule *EXTEND*, to solve conflicts that can arise. This rule simply decides to free a date removing the tuple (“*day*”, d, “*exclusion*”)

Agent

$$Agent = \left\{\!\!\left\{ \begin{array}{l} (\text{“}resume\text{”}), (\text{“}start\text{”} : START), (\text{“}resume\text{”} : RESUME), \\ (\text{”}update\text{”} : U), (\mathbf{terminate} : EXIT), (\text{“}withdraw\text{”} : WITHDRAW), \\ (\text{“}agent\text{”} : AGENT) \end{array} \right\}\!\!\right\}$$

$$START = \left\{\!\!\left\{ \uparrow(\text{“}start\text{”}, me), (\text{“}resume\text{”}) \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \begin{array}{l} (\text{“}done\text{”}), (\text{“}self\text{”}, me), \\ (\text{“}M\text{”}, 1, 0), \ldots, (\text{“}M\text{”}, m, 0) \end{array} \right\}\!\!\right\}$$

$$AGENT = \left\{\!\!\left\{ \begin{array}{l} (\text{“}agent\text{”} : AGENT), \\ ?(\text{“}agent\text{”}) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \mathbf{tsc}(Agent) \right\}\!\!\right\}$$

$$RESUME = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}frozen\text{”}, i, s), \\ (\text{“}resume\text{”}), \uparrow(\text{“}agent\text{”}) \end{array} \right\}\!\!\right\} \xrightarrow{(d_1,\ldots,d_m) \leftarrow f(s)} \left\{\!\!\left\{ \begin{array}{l} (\text{“}M\text{”}, 1, d_1), \ldots, \\ (\text{“}M\text{”}, m, d_m), \\ (\text{“}self\text{”}, i), \uparrow(\text{“}go\text{”}) \end{array} \right\}\!\!\right\}$$

where $f(x) = (unzip_1(x), \ldots, unzip_m(x))$

$$U = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}day\text{”}, 1, d_1), \ldots, \\ \uparrow(\text{“}day\text{”}, m, d_m), \\ (\text{“}M\text{”}, 1, v_1), \ldots, \\ (\text{“}M\text{”}, m, v_m), \\ ?(\text{“}self\text{”}, me) \end{array} \right\}\!\!\right\} \xrightarrow{(\overline{e},\overline{w}) \leftarrow f(\overline{d},\overline{v},me)} \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}day\text{”}, 1, e_1), \ldots, \\ \uparrow(\text{“}day\text{”}, m, e_m), \\ (\text{“}M\text{”}, 1, w_1), \ldots, \\ (\text{“}M\text{”}, m, w_m), \\ (\text{“}done\text{”}) \end{array} \right\}\!\!\right\}$$

where $f(\overline{x}, \overline{y}, z) = (if\ (\forall j x_j \neq z \wedge k = min\{j | x_j = \text{“}free\text{”}\}) then\ (\overline{x}_{|x_k=z}, \overline{y}_{|y_k=y_k+1})$
$else\ (\overline{x}, \overline{y}))$

$$WITHDRAW = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}day\text{”}, h, me), (\text{“}M\text{”}, h, n_1), \\ ?(\text{“}self\text{”}, me), \mathbf{ask}(n_1 > 0) \end{array} \right\}\!\!\right\} \xrightarrow{(n_2) \leftarrow f(n_1)} \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}day\text{”}, h, \text{“}free\text{”}), \\ (\text{“}M\text{”}, h, n_2), (\text{“}done\text{”}) \end{array} \right\}\!\!\right\}$$

where $f(x) = (x - 1)$

$$EXIT = \left\{\!\!\left\{ \begin{array}{l} ?(\text{“}M\text{”}, 1, v_1), \ldots, \\ ?(\text{“}M\text{”}, m, v_m), \\ ?(\text{“}done\text{”}), (\text{“}self\text{”}, i) \end{array} \right\}\!\!\right\} \xrightarrow{(s) \leftarrow f(v_1,\ldots,v_m)} \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}frozen\text{”}, i, s), \\ \uparrow(\text{“}agent\text{”} : AGENT) \end{array} \right\}\!\!\right\}$$

where $f(x_1, \ldots, x_m) = (zip(x_1, \ldots, x_m))$

Table 4.5: The Meeting Scheduler: the Agent Space.

and emitting ("*day*", d, "*free*").

The *Agent* space (Table 4.5) contains some rules and a termination rule to make the agent to freeze. The rule *START* fires an agent to build a calendar (i.e. the tuples ("M", d, v) where d is a day and v is the number of potential attendees for day d, initially all days are free then for all these tuples v is 0. The rule *AGENT* generates (by **tsc**) a new agent space inside the participant space (Fig. 4.4). The first rule enabled in a new agent space, inside a participant space, is *RESUME*: this rule is used to get and restore the frozen state of the agent. It emits a tuple ("*go*") enabling rule *PUSHAG* for a next move. An agent contains also rule U (Update) and rule *WITHDRAW*. The rule U updates the agenda of a participant using the following policy: a participant takes the first free date, if it exists, and books it; a participant cannot book more than one date. Rule U also updates the internal agent table[1], represented by tuples like ("M", d, v) as explained above. In Fig. 4.4 an updating is shown: the participant agenda is updated booking day "1" with the name of the meeting (i.e. the name of the agent): "Z", and increasing by 1 the counter of the meeting potential attendees for day "1" (that now is 2) in the Agent Table. The rule *WITHDRAW* models a withdrawing from a meeting by a participant. It consumes the tuple ("*day*", h, *me*) and emits a tuple ("*day*", h, *free*) in the *Participant* space. It also decreases the number of supposed participants to the meeting h (i.e., it consumes the tuple ("M", h, n_1) and emits the tuple ("M", h, n_2) where $n_2 = n_1 - 1$. The rule *EXIT* is a *termination* rule (see Section 2.1 for its semantics). It terminates the agent space, by freezing the agent and moving it outside: this is performed producing a tuple that represents the frozen state ("*frozen*", i, s) and a tuple ("*agent*" : *AGENT*) for regenerating an *Agent* space.

The model of mobility of the meeting agents is exactly the one specified in Figure 4.2. The meeting agent space is frozen when the agent has finished collecting informations in the participant site. Then, some tuples are emitted in the main space and other participants can get the agent tuples and the agent is regenerated inside another participant site. When the meeting agent has finished, a tuple with the decided meeting date is emitted and the agent is destroyed (by the rule *END*).

[1]Each agent tries to establish a single meeting and the table contains, for each date, the number of participants that would accept that date.

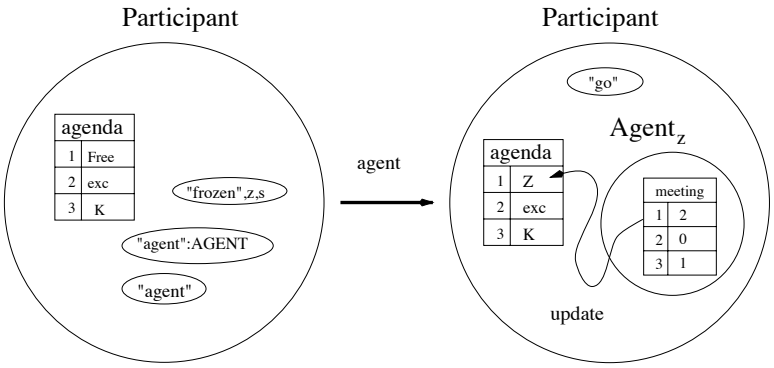

Figure 4.4: Agent performing update.

4.1.3 Analysis of the Meeting Scheduler System

We have used PoliMC to analyze some liveness properties of this system. We use the model checker on finite versions of the specification. The properties we prove for these versions are satisfiable on the abstract specification but we cannot say that such properties are verified in general. Since some components can move we are interested in studying the dynamic behavior of the system. For instance, we would like to prove that an agent will be able to establish a meeting date, or that some properties on the migration of an agent inside/outside the components are true. Formally we can write:

$$End = (\forall agent(\exists day((\text{“}end\text{”}, day, agent) \in StartContext)))$$

$$Move = (((\text{“}done\text{”}) \in \&Agent) \in \&Participant)$$

End states that each agent finds a date for its meeting (i.e., all the meetings are arranged). *Move* states that an agent is in a participant site (i.e., an agent space is inside a participant space) and it has performed some actions (i.e., the tuple (“*done*”) is produced). We study a configuration where the number of meetings to be arranged, namely the number of agents, is smaller than the available days, otherwise, trivially, some agents will never find a date. We would like to verify the following:

$$\star\Box\star\Diamond End \tag{4.1}$$

That is: the *End* property (i.e., each agent finds a date for its meeting) will be valid for all the execution (i.e., all the execution paths lead to a state where the *End* condition is

verified). However, PoliMC shows that (4.1) is false. To understand this we can think of a scenario where agents are not able to agree, choosing the same date and then withdrawing it. Nevertheless, PoliMC also verifies the falsity of:

$$\star\Box\star\Diamond Move \tag{4.2}$$

Property (4.2) states that the *Move* property will be valid (i.e. agents move indefinitely) for all executions. The falsity of (4.2), verified with the model checker, guarantees that this cannot happen, so we are sure to have a scenario where all meetings are arranged. PoliMC verifies that this property is not true if the number of meetings (agents) is greater than the number of the available days.

As (1) and (2) are proved false we can verify the following formula:

$$\star\Box\star\Diamond(End \vee Move) \tag{4.3}$$

That is, in all the executions it is true that some agents move or all the meetings are arranged. This shows that the system cannot deadlock. In this example however, properties (4.1) and (4.2) above cannot help us to guarantee progress. Therefore, to ensure that all the meetings will be arranged we need a fairness condition in the form:

$$\star\Box\&\Diamond End \Rightarrow (4.1) \tag{4.4}$$

that is:

$$\star\Box\&\Diamond End \Rightarrow \star\Box\star\Diamond End \tag{4.5}$$

Property 4.5 states that if from all the states of all paths we can find at least one path in which *End* is eventually valid, then *End* will be valid in all the paths. In other words if we are always in a state that allows to arrange all the meetings, then this will eventually happen. PoliMC verifies successfully the hypothesis of Property 4.5:

$$\star\Box\&\Diamond End \tag{4.6}$$

hence, Property (4.6) in conjunction with (4.5) leads to the verification of (4.1).

Finally we remark that if we remove some rules used to resolve conflicts (like rule *WITHDRAW* or rule *EXTEND*), (4.6) is not verified, that is, there are some states where

no path leads to *End*. In other words, sometimes a system can reach a state in which it is impossible to arrange some meetings, and some agents move indefinitely. To avoid these situations the withdrawing or the extension of the free dates by some participants should be considered.

In order to verify properties using the model checker we have instantiated the specification of the Meeting Scheduler System. Here we show a part of the specification instantiated with two possible meeting days (i.e. two agents), and two participants. The model checker accepts as input two files containing respectively the PoliS specification of the system and the formulae to be verified. What follows is a part of the specification file for the Meeting Scheduler:

```
startcontext={
 Participant,Participant,
 Agent,Agent,("start",1),("start",2),
 ("end":END)
}

rule END={
    ("frozen",k,(day1,num1,day2,num2)),
    ask(num1=PART \/ num2=PART)
    }
    [(d)<--f(day1,num1,day2)]-->
    {
     ("end",k,d)
    }
    where f(x,y,z)=(if (y=PART) then (x) else (z));
```

`PART` is a constant defining the number of participants. Notice that the state s of the frozen agent ("*frozen*", k, s) consumed by the rule *END* (Table 4.3) has been expanded in order to express the conditions on the **where** clause, `(day1, num1, day2, num2)`: `day1` and `day2` indicate the two possible meeting dates while `num1` and `num2` indicate respectively the number of the participants to the two meetings.

The check (**ask**) on the expanded state of the meeting agent has also been inserted: it checks if one of the two dates has been chosen by both the participants. Here is the specification of property (4.3):

```
*[]*<> (
(forall agent in [1,2] (
        exists day in [1,2] (
            ("end",day,agent) ))) \/
    ((("done") in & Agent) in & Participant))
```

The range of the agents and the days is explicitly set (`[1,2]`). At the moment the textual specifications input for the model checker have to be written by the specifier, but an interface tool that translates PoliS-LaTeX specifications into textual ones can easily be designed. As the model checker works on finite instances of the specification, the user has to define the range of the parameters. Furthermore, she has to declare the abstract PoliS functions after the **where** clause (i.e., `where f(x,y,z)=(if (y=PART) then (x) else (z))`.

In order to further constrain the state explosion, we are researching techniques of context constrains for compositional reachability analysis (CRA) [GS90, CK96]. As in PoliS the components (namely the spaces) make assumptions on their external environment (namely their parent space) using the rule scope (see Figure 7.1), this kind of analysis can be applied in order to drastically reduce the number of states of the graph. An other approach that could be followed to further reduce the state explosion is symbolic model checking. In [EFT92] a technique for building BDD of parallel processes from basic BDD is exploited. The bottom-up fashion of this approach is similar to our technique of building compound spaces from simple spaces.

Chapter 5

MobiS: an Enhancement of PoliS

PoliS allows the specification of mobile code systems. However, as shown in Chapter 4, agent mobility has to be encoded using rules mobility plus removal and creation of spaces. MobiS is an enhancement of PoliS allowing mobility of agents to be encoded in the language as a first class operation. Agent mobility is encoded as mobility of a "space tuple", that can be consumed and produced as regular tuples. The language therefore allows all ranges of mobility, data, code, and agents to be formalized as basic primitives exploiting the language constructs.

In this chapter we introduce MobiS, the semantics modification needed for enhancing PoliS to MobiS, and some examples.

5.1 MobiS

MobiS is an enhancement of PoliS. In MobiS not only data tuples and program tuples can be contained in a tuple space but also spaces themselves are represented as tuples in the parent spaces. This means that spaces become first class entities and that they can be produced and consumed (i.e. also moved) by rules as regular tuples. When a space is moved to an other place, all the sub-spaces it contains are moved too. This encodes the movement of a component and of its sub-components. MobiS allows us to specify the movement of agents in a software architecture and the reconfigurability of the system.

The following example shows a Client-Server system: the Client and the Server exchange requests and replies. However the architecture is reconfigurable and as the network is supposed to be sometimes busy, the Client sends an Agent to the Server in order to avoid continuous and maybe expensive communication on the link. Then, the Agent and

the Server communicate in the local Server site. When the Agent has finished it goes back to the Client site. MobiS can model the behavior of the mobile agent and the dynamics of the system.

In order to give the idea of how MobiS specifications can be written we show the formalization of the Client component in Table 5.1. The notation looks very much like PoliS except from the new "space tuples".

Client

$$Client = \left\{\!\!\left\{ \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}put\text{”} : PUT), (\text{“}reqlist\text{”}, r), (\text{“}get\text{”} : GET), \\ (\text{“}move\text{”} : MOVE), (\text{“}create\text{”} : CREATE), (\text{“}servername\text{”}, s) \end{array} \right\}\!\!\right\}$$

$$PUT = \left\{\!\!\left\{ ?(\text{“}reqlist\text{”}, r), ?((\text{“}name\text{”}, k), (\text{“}idle\text{”}) \right\}\!\!\right\} \xrightarrow{(t) \leftarrow f(r)} \left\{\!\!\left\{ \uparrow(\text{“}req\text{”}, k, t), (\text{“}wait\text{”}) \right\}\!\!\right\}$$

where $f(x) = (\mathsf{head}(x))$

$$GET = \left\{\!\!\left\{ \begin{array}{l} \uparrow(\text{“}reply\text{”}, i, r), (\text{“}reqlist\text{”}, t), \\ ?(\text{“}name\text{”}, i), (\text{“}wait\text{”}) \end{array} \right\}\!\!\right\} \xrightarrow{(j) \leftarrow f(t)} \left\{\!\!\left\{ \begin{array}{l} (\text{“}requlist\text{”}, j), (\text{“}reply\text{”}, i, r), \\ (\text{“}idle\text{”}) \end{array} \right\}\!\!\right\}$$

where $f(x) = (diff(x, \mathsf{head}(x)))$

$$CREATE = \left\{\!\!\left\{ \uparrow(\text{“}networkbusy\text{”}), ?(\text{“}name\text{”}, i), (\text{“}reqlist\text{”}, r) \right\}\!\!\right\} \xrightarrow{(a) \leftarrow f(r,i)} \left\{\!\!\left\{ (a * Agent) \right\}\!\!\right\}$$

where $f(x, y) = (z)$

$$MOVE = \left\{\!\!\left\{ (a * Agent), ?(\text{“}servername\text{”}, k) \right\}\!\!\right\} \xrightarrow{(j) \leftarrow f(a,k)} \left\{\!\!\left\{ \uparrow(j * Agent) \right\}\!\!\right\}$$

where $f(x, y) = (concat(x, y))$

$$GETAG = \left\{\!\!\left\{ \uparrow(a * Agent), ?(\text{“}name\text{”}, k), \mathbf{ask}(prefix(a, k) \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ (a * Agent) \right\}\!\!\right\}$$

Table 5.1: Specification of the Client component

The Client space contains an ordinary tuple indicating the name of the client (“*name*”, k) where k is the formal parameter containing the name. It also contains the tuple (“*reqlist*”, r) of the list of the requests for the server, the name of the server (“*servername*”, s), and some program tuples that refer to rules specified below in the table.

The rules *PUT* and *GET* handle the communication with the server when the network is not busy. The rule *PUT* emits in the external space (the network) a request extracting it from the requests list. The rule *GET* gets the reply from the Server (i.e. it checks if

a reply directed to the Client is present on the network), and stores it in the local space updating the requests list (it throws the first request in the queue as it has already been served).

When the network is busy the rule *CREATE* generates an *Agent* space storing the requests in it.

The difference with PoliS can be seen at this point. Space tuples are represented with the "*". The symbol is put between the name of the space and the type of it. The type represents the list of things in the space. For instance, the space tuple ("$a'' * Agent$) represents a space named a with content represented by the type *Agent* (that for brevity we do not define here but that looks similar to the definition of the type *Client*).

The rule *MOVE* moves the Agent into the network. It also changes the name of the Agent appending the name of the Server to it in order to indicate the destination of the Agent.

The last rule is *GETAG* that gets the Agent from the network when it come back after having finished its work on the Server site.

The Client can choose the communication protocol depending on its context: when the network is not congestioned it sends requests and wait for replies, while when the network becomes busy it builds an Agent and sends it to the Server site to exploit local computation.

5.2 The semantics and the difference with PoliS

We now define the semantics of MobiS. As the model is largely derived from PoliS we will make references to the tables shown in Chapter 2 containing the semantics of PoliS and illustrate the differences.

The differences with PoliS

MobiS spaces are represented as tuples in the form ($spacename * S$), where *spacename* is the space name and S is the shortcut for the the contained space. Every space contains a mandatory tuple ("$name'', n$), where n is the space name and is semantically bound to *spacename*. In PoliS spaces do not have names. The names on top of the spaces definition tables are only the shortcut used for the substitution in the code.

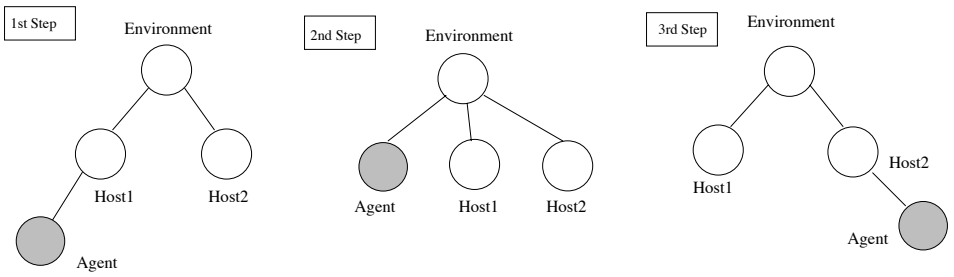

Figure 5.1: The movement of a space.

With the additional space names, nothing prevents the shipping of space names across the network allowing remote spaces to have knowledge on what is here.

PoliS operators such as **tsc** (tuple space creation) and **terminate** (space termination) are not necessary anymore as spaces can be consumed and produced in the same way as regular tuples are. MobiS spaces therefore can be mobile as code was mobile in PoliS. Figure 5.1 shows the idea.

A space (that we can call an agent) can be moved from one host to another is three steps. PoliS and MobiS rules have a specific scope that drives the movement of tuples (Figure 7.1). The first step of the movement of a space that needs to be transferred from Host1 to Host2 is to be moved up in the common environment and then be picked up by a rule in Host2. We already showed a small example of MobiS specification. In Section 5.3 we show a more complex example.

MobiS Semantics

The differences outlined in the previous section between PoliS and MobiS lead to a set of modification both in syntax and in the semantics. In Table 5.2 we show the syntax of MobiS. The main syntactic difference with PoliS is the introduction of space tuples and the mandatory binding of the name to a tuple name inside the space.

Table 5.3 show the modified rules with respect to Table 2.3. The **terminate** rule does not exist in MobiS and the space are added and consumed as the other tuples.

In Table 5.4 the **terminate** rule is not shown unlike in Table 2.4 as it does not exist anymore and the **tsc** operators to generate space are abolished.

Table 5.5 shows the enabling conditions for the rules defined in Table 5.4. The condition check that the name of the spaces are not consumed.

MS	::=	$\{\!\!\{$ tuple $\}\!\!\}$ | MS $\oplus$ MS | MS $\setminus$ MS | (MS)
tuple	::=	data | program | space
space	::=	(*sname** ((“*name*”,*sname*) $\oplus$ MS)
program	::=	(*rname* : *Code*)
data	::=	(datalist)
datalist	::=	*data* | *value* | *data*, datalist | *value*, datalist

sname $\in$ *Spaceid*, the set of space identifiers

rname $\in$ *Ruleid*, the set of rule identifiers

Code $\in$ *Rulecode*, the set of rules code specified in Table 5.4.

In the concrete syntax, *Code* is usually substituted with a macro that expands in the code itself.

value $\in$ *Values*

data $\in$ *String*

Table 5.2: MobiS Abstract Syntax.

Local Rule

RL: $\{\!\!\{(\text{“}r_l\text{”} : R_l)\}\!\!\} \oplus M \longrightarrow ((\{\!\!\{(\text{“}r_l\text{”} : R_l)\}\!\!\} \oplus M) \setminus \{\!\!\{\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\}) \oplus \{\!\!\{\overline{t}_p[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]\}\!\!\}$

if $LocEnabled(R_l, \text{“}r_l\text{”}, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}})$

Interaction Rule

RI: $\{\!\!\{\{\!\!\{(\text{“}r_i\text{”} : R_i)\}\!\!\} \oplus M_1\}\!\!\} \oplus M_2 \longrightarrow \{\!\!\{((\{\!\!\{(\text{“}r_i\text{”} : R_i)\}\!\!\} \oplus M_1) \setminus \{\!\!\{\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\}) \oplus \{\!\!\{\overline{t}_p[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]\}\!\!\}\}\!\!\}$
$\oplus (M_2 \setminus \{\!\!\{\overline{t}_{ec}[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\}) \oplus \{\!\!\{\overline{t}_{ep}[\overline{v}_{\overline{x}}/\overline{x}, \overline{v}_{\overline{y}}/\overline{y}]\}\!\!\}$

if $IntEnabled(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}})$

Table 5.3: MobiS Structured Operational Semantics Rules and Axioms.

5.3 Using MobiS for Agent Mobility across a Network

In this section we show how MobiS can be used to specify a scenario where agents move over a network. The hierarchical organization of spaces in MobiS reflects a real network organization, composed of layered domains. Figure 5.2 shows the structure of the network.

The main space we consider is a WAN and it is an abstraction of a wide area network in which LANs (local area networks) are contained. The LANs are composed of many Hosts that represent the different sites. We can imagine that different agents move on the

$$R_l = \left\{\!\!\left\{ \begin{array}{l} t_{c,1},\ldots,t_{c,n_c},?t_{t,1},\ldots,?t_{t,n_t}, \\ \textbf{ask}(boolexpr) \end{array} \right\}\!\!\right\} \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \left\{\!\!\left\{ t_{p,1},\ldots,t_{p,n_p} \right\}\!\!\right\}$$

where $f((\overline{z})) = ((f_1(\overline{z}),\ldots,f_m(\overline{z})))$

$$R_i = \left\{\!\!\left\{ \begin{array}{l} t_{c,1},\ldots,t_{c,n_c},\uparrow t_{ec,1},\ldots,\uparrow t_{ec,n_{ec}}, \\ ?t_{t,1},\ldots,?t_{t,n_t},?\uparrow t_{et,1},\ldots,?\uparrow t_{et,n_{et}}, \\ \textbf{ask}(boolexpr) \end{array} \right\}\!\!\right\} \xrightarrow{(\overline{y})\leftarrow f(\overline{x})} \left\{\!\!\left\{ \begin{array}{l} t_{p,1},\ldots,t_{p,n_p}, \\ \uparrow t_{ep,1},\ldots,\uparrow t_{ep,n_{ep}} \end{array} \right\}\!\!\right\}$$

where $f((\overline{z})) = ((f_1(\overline{z}),\ldots,f_m(\overline{z})))$

Table 5.4: Classification of PoliS Rules Macro.

$$\begin{array}{l} LocEnabled(R_l, \text{“}r_l\text{”}, M, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \triangleq \\ \quad \{\!\!\{\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_t[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\} \subseteq \{\!\!\{(\text{“}r_l\text{”} : R_l)\}\!\!\} \oplus M \\ \quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge boolexpr[\overline{v}_{\overline{x}}/\overline{x}] \wedge \forall n : (\text{“}name\text{”}, n) \notin \overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}] \\ IntEnabled(R_i, \text{“}r_i\text{”}, M_1, M_2, \overline{v}_{\overline{x}}, \overline{v}_{\overline{y}}) \triangleq \\ \quad \{\!\!\{\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_t[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\} \subseteq \{\!\!\{(\text{“}r_i\text{”} : R_i)\}\!\!\} \oplus M_1 \wedge \{\!\!\{\overline{t}_{ec}[\overline{v}_{\overline{x}}/\overline{x}], \overline{t}_{et}[\overline{v}_{\overline{x}}/\overline{x}]\}\!\!\} \subseteq M_2 \\ \quad \wedge\ \overline{v}_{\overline{y}} = f(\overline{v}_{\overline{x}}) \wedge boolexpr[\overline{v}_{\overline{x}}/\overline{x}] \wedge \forall n : (\text{“}name\text{”}, n) \notin (\overline{t}_c[\overline{v}_{\overline{x}}/\overline{x}] \vee\ \overline{t}_{ec}[\overline{v}_{\overline{x}}/\overline{x}]) \end{array}$$

Table 5.5: Precondition predicates.

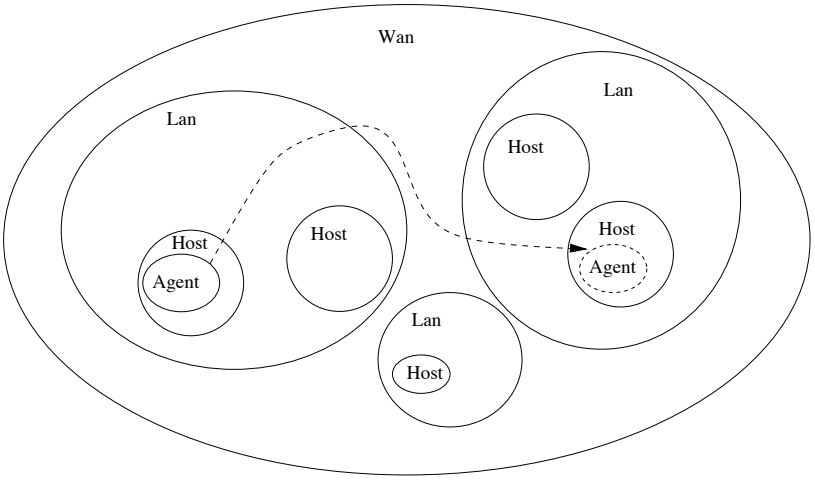

Figure 5.2: The Network.

network from host to host, not necessarily in the same LAN.

Hosts can generate agents and send them over the network in order to perform computations remotely. We now describe in details the MobiS specification contained in Tables 5.6, 5.7, and 5.8. Table 5.6 contains the specification of a wide area network (WAN): it

$$WAN = \{\!|\ (\text{“}name\text{”}, id), (\text{“}l1\text{”} * LAN), \ldots, (\text{“}ln\text{”} * LAN)\ |\!\}$$

$$LAN = \left\{\!\!\left|\begin{array}{l}(\text{“}name\text{”}, id), (\text{“}h1\text{”} * HOST), \ldots, (\text{“}hm\text{”} * HOST),\\ (\text{“}g\text{”} : GETAG), (\text{“}m\text{”} : MOVEAG)\end{array}\right|\!\!\right\}$$

$$GETAG = \left\{\!\!\left|\begin{array}{l}\uparrow(a * AGENT), (\text{“}name\text{”}, k),\\ \textbf{ask}(prefix(k, a))\end{array}\right|\!\!\right\} \longrightarrow \{\!|\ (a * AGENT)\ |\!\}$$

$$MOVEAG = \left\{\!\!\left|\begin{array}{l}(a * AGENT), (\text{“}name\text{”}, k),\\ \textbf{ask}(\neg prefix(k, a))\end{array}\right|\!\!\right\} \longrightarrow \{\!|\ \uparrow(a * AGENT)\ |\!\}$$

Table 5.6: Specification of the Network with Agents System: the WAN and LAN spaces.

contains some spaces tuples that refer to space LAN ($(li * LAN)$). It also contains its name (i.e a domain identifier).

A *LAN* space contains its identifier, some hosts ($(hi * HOST)$), and some program tuples referring to the following rules. The rule *GETAG* gets an agent from the WAN space if the destination address of the agent is a host in its domain. The check is performed considering the names of the spaces as hierarchically structured: we check if the LAN ID is a prefix of the destination name. The rule *MOVEAG* moves the agent space out of the LAN space in case the destination of the agent is not in the LAN domain.

Table 5.7 contains the specification of the *HOST* space. The *HOST* space contains an identifier tuple ($(\text{“}name\text{”}, id)$), and a program tuple that refers to a rule able to generate agents ($(\text{“}g\text{”} : GENERATE)$). It also contains the tuple ($(\text{“}resources\text{”}, r)$) indicating the resources of the host. The rule *GENERATE*, when applied generates a new agent in the local space. It tests the identifier of the host ($?(\text{“}name\text{”}, id)$) and generates a space tuple ($a * AGENT$) that refers to the specification of the space *AGENT* shown in table 5.8. The name a of the agent space refers to the host ID with an appended random ID (it is computed by the function specified on the rule arrow and defined after the clause **where**). The rule *RECOGN* gets the agent id-tuple emitted by the agent (see table 5.8), performs

$$HOST$$

$$HOST = \left\{\!\!\left\{ \begin{array}{l} (\text{“}name\text{”}, id), (\text{“}g\text{”} : GENERATE), (\text{“}resources\text{”}, r), \\ (\text{“}rec\text{”} : RECOGN), (\text{”}kill\text{”} : KILL), \\ (\text{“}get\text{”} : GET), (\text{”}m\text{”} : MOVE) \end{array} \right\}\!\!\right\}$$

$$GENERATE = \left\{\!\!\left\{ ?(\text{“}name\text{”}, n) \right\}\!\!\right\} \xrightarrow{(a) \leftarrow f(n)} \left\{\!\!\left\{ (a * AGENT) \right\}\!\!\right\}$$

where $f(i) = (concat(i, random(i)))$

$$RECOGN = \left\{\!\!\left\{ (\text{“}idagent\text{”}, k), \textbf{ask}((k \leq 100) \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ (\text{“}go\text{”}) \right\}\!\!\right\}$$

$$KILL = \left\{\!\!\left\{ (\text{“}idagent\text{”}, k), \textbf{ask}((k > 100), (a * AGENT) \right\}\!\!\right\} \longrightarrow \{\!\!\{\}\!\!\}$$

$$GET = \left\{\!\!\left\{ \begin{array}{l} \uparrow(a * AGENT), (\text{“}name\text{”}, k), \\ \textbf{ask}(k = a) \end{array} \right\}\!\!\right\} \xrightarrow{(b) \leftarrow f(k)} \left\{\!\!\left\{ (b * AGENT) \right\}\!\!\right\}$$

where $f(id) = (concat(id, random(id)))$

$$MOVE = \left\{\!\!\left\{ \begin{array}{l} (a * AGENT), (\text{“}dest\text{”}, addr), \\ (\text{“}move\text{”}) \end{array} \right\}\!\!\right\} \longrightarrow \left\{\!\!\left\{ \uparrow(addr * AGENT) \right\}\!\!\right\}$$

Table 5.7: Specification of the Network with Agents System: the Host Space.

some checks (we only check that the tuple-id is a number smaller than 100), and emits the tuple (“*go*”). This rule is used whenever an agent arrives to the destination host and has to be authorized to execute. If the agent tuple-id emitted is a number larger than 100 the rule *KILL* kills the unauthorized agent.

The rule *MOVE*, when enabled (by the tuple (“*move*”) put in the local space by the agent), moves the agent out of the host space. It also changes the agent space name into its destination address.

The rule *GET* simply gets the agent when it corresponds to its destination space consuming it from the parent space and producing it locally. It checks by the **ask** clause if the name of the agent is its name.

The *AGENT* space contains a name tuple, the code, the state and the store. The rule *EXE* performs some computations emitting a result tuple. It is enabled when the authorization phase in the destination site is finished: the tuple (“*go*”) is emitted by the Host.

$$
\begin{array}{l}
\textit{AGENT} \\
\textit{AGENT} = \left\{ \begin{array}{l} (\text{“}name\text{”}, k), (\text{“}idle\text{”}), (\text{“}code\text{”}, c), (\text{“}state\text{”}, s), (\text{“}store\text{”}, st), \\ (\text{“}r\text{”} : READY), (\text{“}exe\text{”} : EXE), (\text{“}aut\text{”} : AUTHORIZE) \end{array} \right\} \\
\\
READY = \left\{ \begin{array}{l} ?\uparrow(\text{“}dest\text{”}, addr), \\ (\text{“}idle\text{”}) \end{array} \right\} \longrightarrow \left\{ \begin{array}{l} \uparrow(\text{“}move\text{”}), (\text{“}name\text{”}, addr), \\ (\text{“}moving\text{”}) \end{array} \right\} \\
\\
AUTHORIZE = \left\{ \begin{array}{l} (\text{“}arrived\text{”}), \\ (\text{“}name\text{”}, k) \end{array} \right\} \xrightarrow{(j) \leftarrow f(k)} \left\{ \uparrow(\text{“}idagent\text{”}, j) \right\} \\
\textbf{where } f(k) = (get_i\, d(j)) \\
\\
EXE = \left\{ \begin{array}{l} \uparrow(\text{“}go\text{”}), (\text{“}code\text{”}, c), \\ (\text{“}state\text{”}, s), (\text{“}store\text{”}, st), \\ \uparrow(\text{“}resources\text{”}, r) \end{array} \right\} \xrightarrow{(res) \leftarrow f(c,s,st,r)} \left\{ (\text{“}result\text{”}, res) \right\} \\
\textbf{where } f(cod, state, store, resou) = (result)
\end{array}
$$

Table 5.8: Specification of the Network with Agents System: the Agent space.

The *AUTHORIZE* rule emits the id-tuple of the agent in the host space letting the rule *RECOGN* of the host to do its job.

When the agent is ready to move it gets its destination address from the host space and emits in the host space the tuple (“*move*”), enabling the rule *MOVE*, by the rule *READY*. This rule also changes the state of the agent from *idle* to *moving*, and its name to the name of the destination.

In the specification shown the agent is generated by an host, that also assigns to it a destination (a site to reach). The mobility imposed by the model is “controlled”: when the agent wants to move it signals this intention to its host that moves the agent space out of its site. The exit from a LAN (when the destination host is in an other domain), is specified in a similar way: a rule of the LAN space expels the agent when its destination address is not one of the hosts addresses contained in the LAN. When an agent is in the general WAN, special rules in the contained LAN spaces check for agents in the WAN with destination address members of their domains. If they find these kind of agents, they move them in their local spaces. The host itself does the same: when an agent

destined to its site is found in the LAN it moves it into its local space. This controlled mobility can simulate the autonomous mobility where the object simply moves, exploiting the mechanisms of synchronizations among rules. By the way, controlled mobility also allows an host to move an object without its willing (it is not the case of our specification but it could be possible).

We think that many security aspects can be analyzed on the basis of this kind of controlled mobility. The fact that a parent space cannot look into its sub-spaces is, on one hand an advantage: an object inside an host is secure, the host cannot, for instance, modify its code. On the other hand, the agent can lie to the host and get an authorization to execute exploiting and damaging the resources of the host.

5.4 Specification of Architectural Styles for Mobility

MobiS ability to specify data, code and agent mobility as first class allows the formal definition of different mobility paradigms that can be reused in the design of applications. In this section we show the formalization of the paradigms and the use of them to define an architecture.

An architectural style is an abstract skeleton which helps in designing, understanding, and analyzing actual software architectures, said *instances* of such a style.

There are at least three reasons why it is important and useful to systematically study architectural styles:

- to help designers to choose a specific style in a given design situation; the definition and classification of common architectural styles with clearly defined properties supports both design and code reuse;
- to build a library of styles, so that software designers can choose the most appropriate one;
- to support analysis methods and tools suitable to deal with style instances, namely concrete software architectures, understanding and reasoning on their properties.

We have defined a basic set of architectural styles for mobility. We catalog these *mobile architectural styles* in terms of *what is moving*, namely which entities move with respect to an infrastructure including at least two immobile entities: a *requester* entity and a *supplier* entity. The requester asks the supplier for a service, and the supplier provides the service.

These two entities are actually part of the styles as they characterize the structure of the environment. Both these immobile entities can be thought of as two Internet sites connected by some channel able to transport mobile entities from a site to another.

1. **Data Style**: This is the simplest kind of mobility to understand. The mobile entities are data from a supplier to a requester. A typical example is a client-server architecture based on a protocol like HTTP: HTTP servers send to HTTP clients data in form of HTML pages (HTML being a not Turing complete language).

2. **Code Style**: in this case some executable code can move from a site to an other. Java applets are based on code mobility.

3. **Ambient Style**: this style describes the moving of the whole ambient involved in a computation. Ambients can contain other ambients that are moved too. In this way it is possible to model, for instance, the moving of a set of programs from a workstation to a laptop. At the moment no languages exist allowing this kind of mobility. However, Cardelli and Gordon have proposed a programming language based on this paradigm [CG98].

MobiS allows the specification of architectural styles for mobility. Architectural styles are abstractions including components, connectors [SG96]. Components are computation loci, while connectors define the interactions among components. In MobiS components can be specified as spaces (that can also be nested): new components can be generated (i.e. spaces can be created), eliminated (i.e. spaces can be consumed), or can migrate (i.e. spaces can be consumed and recreated elsewhere). The way in which MobiS models software architectures [CM98b] is similar to the one described in [IW95] where the CHAM coordination model is used. The coordination allows flexible moving of components and extensibility of the model. Our model, where rules and spaces as first class entities, provides a framework in which encoding all the different styles listed above. The concept of connector in MobiS is in some sense implicit (as in [MK96, IW95]): components interactions are defined by the coordination model, whereas communication is specified using the asynchronous mechanism of multiset rewriting.

We now specify some mobility styles using MobiS. Table 5.10 contains the specification of the **Data style**. The main space contains two spaces, a "Requester" and a "Supplier".

Environment

$$Environment = \{\!|\ (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier)\ |\!\}$$

Requester

$$Requester = \left\{\!\!\left|\begin{array}{l}(\text{“}name\text{”}, k), (\text{“}codereq\text{”}, req), (\text{“}request\text{”} : CODEREQ),\\ (\text{“}get\text{”} : GET), (\text{“}store\text{”}, st), (\text{“}state\text{”}, s)\end{array}\right|\!\!\right\}$$

$$CODEREQ = \{\!|\ ?(\text{“}codereq\text{”}, req)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}codereq\text{”}, req)\ |\!\}$$

$$GET = \{\!|\ \uparrow(\text{“}serializedcode\text{”}, sc), (\text{“}codereq\text{”}, req)\ |\!\} \xrightarrow{(c) \leftarrow f(sc)} \{\!|\ (\text{“}code\text{”}, c)\ |\!\}$$

where $f(x) = (f_{code}(x))$

Supplier

$$Supplier = \{\!|\ (\text{“}name\text{”}, k), (\text{“}c\text{”} : CODE), (\text{“}put\text{”} : PUT), (\text{“}getreq\text{”} : GETREQ)\ |\!\}$$

$$GETREQ = \{\!|\ \uparrow(\text{“}codereq\text{”}, req), ?(\text{“}code\text{”}, c)\ |\!\} \longrightarrow \{\!|\ (\text{“}codereq\text{”}, req)\ |\!\}$$

$$PUT = \{\!|\ (\text{“}codereq\text{”}, req), ?(\text{“}code\text{”}, c)\ |\!\} \xrightarrow{(sc) \leftarrow f(c)} \{\!|\ \uparrow(\text{“}serializedcode\text{”}, sc)\ |\!\}$$

where $f(x) = (serialize(x))$

$$CODE = \left\{\!\!\left|\begin{array}{l}?(\text{“}store\text{”}, st),\\ ?(\text{“}state\text{”}, s),\end{array}\right|\!\!\right\} \xrightarrow{(s', st) \leftarrow f(s, st)} \left\{\!\!\left|\begin{array}{l}(\text{“}state\text{”}, s'),\\ (\text{“}store\text{”}, st')\end{array}\right|\!\!\right\}$$

where $f(store, state) = (nstate(store, state))$

Table 5.9: MobiS specification of Code Paradigm

The *Requester* space contains a data reference request tuple, the code, and the state. The rule *DATAREQ* sends a data request to the main space. The rule *GET* gets from the main space the data. The rule *CODE* formalizes the execution of the requester code using the data. Notice that as the spaces are organized in a tree it is quite easy to specify the access to the data: spaces have names in form of paths.

Table 5.9 contains the formalization of the **Code style**: The *Requester* space contains a code-request tuple, the state, and the store. The rule *CODEREQ* sends a code request

Environment

$$Environment = \{\!|\ (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier)\ |\!\}$$

Requester

$$Requester = \left\{\!\!\left|\ \begin{array}{l}(\text{“}name\text{”}, k), (\text{“}datareq\text{”}, req), (\text{“}request\text{”} : DATAREQ),\\ (\text{“}get\text{”} : GET)(\text{“}code\text{”} : CODE), (\text{“}data\text{”}, d), (\text{“}state\text{”}, s)\end{array}\ \right|\!\!\right\}$$

$$DATAREQ = \{\!|\ ?(\text{“}datareq\text{”}, req)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}datareq\text{”}, req)\ |\!\}$$

$$GET = \{\!|\ \uparrow(\text{“}data\text{”}, rd), (\text{“}datareq\text{”}, req)\ |\!\} \longrightarrow \{\!|\ (\text{“}data\text{”}, rd)\ |\!\}$$

$$CODE = \{\!|\ ?(\text{“}data\text{”}, rd), ?(\text{“}state\text{”}, s),\ |\!\} \xrightarrow{(s') \leftarrow f(s, rd)} \{\!|\ (\text{“}state\text{”}, s')\ |\!\}$$

where $f(data, state) = (nstate(data, state))$

Supplier

$$Supplier = \{\!|\ (\text{“}name\text{”}, k), (\text{“}data\text{”}, d), (\text{“}put\text{”} : PUT), (\text{“}getreq\text{”} : GETREQ)\ |\!\}$$

$$GETREQ = \{\!|\ \uparrow(\text{“}datareq\text{”}, req), ?(\text{“}data\text{”}, d)\ |\!\} \longrightarrow \{\!|\ (\text{“}datareq\text{”}, req)\ |\!\}$$

$$PUT = \{\!|\ (\text{“}datareq\text{”}, req), ?(\text{“}data\text{”}, d)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}data\text{”}, d)\ |\!\}$$

Table 5.10: MobiS specification of the Data Paradigm.

in the main space. The rule *GET* gets from the main space the serialized code sent by the supplier. The rule *CODE* formalizes the execution of code updating the values of the state and the store. The *Supplier* space contains the code and two rules. The rule *GETREQ* accepts a request of code from the main space, and the rule *PUT* emits the serialized code in the main space.

Table 5.11 contains the specification of the **Ambient style.** The *Supplier* space contains two rules and an *Ambient* subspace. The rule *PUT* transfers the ambient space outside. It changes the location of the ambient as in the Closure style. The *Ambient* space contains the code, the state, the store, and some resources.The rule *CODE* executes the

Environment

$$Environment = \{\!|\ (\text{“}R\text{”} * Requester), (\text{“}S\text{”} * Supplier)\ |\!\}$$

Supplier

$$Supplier = \{\!|\ (a * Ambient), (\text{“}name\text{”}, k)(\text{“}put\text{”} : PUT)\ |\!\}$$

$$PUT = \left\{\!\!\left|\begin{array}{l} \uparrow(\text{“}req\text{”}, req), ?(\text{“}name\text{”}, i), \\ (a * Ambient) \end{array}\right|\!\!\right\} \xrightarrow{(z) \leftarrow f(a,i)} \{\!|\ \uparrow(z * Ambient)\ |\!\}$$

where $f(x, y) = (diff(x, y))$

Ambient

$$Ambient = \left\{\!\!\left|\begin{array}{l} (\text{“}name\text{”}, k), (\text{“}data\text{”}, d), (\text{“}state\text{”}, s), \\ (\text{“}c\text{”} : CODE), (\text{“}resources\text{”}, r) \end{array}\right|\!\!\right\}$$

$$CODE = \left\{\!\!\left|\begin{array}{l} ?(\text{“}store\text{”}, st), \\ ?(\text{“}state\text{”}, s), (\text{“}resource\text{”}, r) \end{array}\right|\!\!\right\} \xrightarrow{(s', st') \leftarrow f(r,s,st)} \left\{\!\!\left|\begin{array}{l} (\text{“}store\text{”}, st'), \\ (\text{“}state\text{”}, s') \end{array}\right|\!\!\right\}$$

where $f(res, store, state) = (nstate(res, store, state), nstore(res, store, state))$

Requester

$$Requester = \{\!|\ (\text{“}req\text{”}, req), (\text{“}request\text{”} : REQ), (\text{“}get\text{”} : GET), (\text{“}name\text{”}, k)\ |\!\}$$

$$REQ = \{\!|\ ?(\text{“}req\text{”}, req)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}req\text{”}, req)\ |\!\}$$

$$GET = \left\{\!\!\left|\begin{array}{l} \uparrow(a * Ambient), (\text{“}req\text{”}, req), \\ ?(\text{“}name\text{”}, z) \end{array}\right|\!\!\right\} \xrightarrow{(j) \leftarrow f(z,a)} \{\!|\ (j * Ambient)\ |\!\}$$

where $f(x, y) = (concat(x, y))$

Table 5.11: MobiS specification of the Ambient style

code using the local resources to the ambient, no matter where the ambient is located. The *Requester* rule *GET* gets the *Ambient* space from the parent space and updates its

location name. This last paradigm can be used to model agent mobility where the agent is identified with an ambient.

5.5 Application of the Styles to the Architecture of a Mobile System

We consider the software architecture of a system and apply these styles to see how these paradigms can be used.

As a case study we consider an electronic commerce application. With the advancements in the network technology new kinds of applications are now possible. A purchaser is trying to buy items at the best available prices on the network. The purchaser travels on the network looking for the best selling-price. We have simplified the problem supposing that the purchaser is looking for the best price of a single object. We exploit the mobility styles defined in Section 5.4 to specify the software architecture of the *Purchasing System*.

Using the **Data style** the purchaser can be seen as a requester that asks for the items prices from different stores. The stores send prices of the items to the purchaser that can remotely check the prices and choose the lower one. The purchaser still does not move, and it can remotely check the prices on the catalogs. In the **Code style** solution we imagine the purchaser migrating from a store site to an other moving its code. Every store puts an advertisement request tuple, ("*newsellingprice*", *reqselling*), in the main space. The store containing the code of the purchasing agent emits the code tuple in the main space and the store that puts the advertisement can obtain the purchasing code. However this solution is not suitable for the purchasing system, because the purchaser has to remember the best price found every time it moves. This solution fits better the purchasing system than the one with the **Code style**, in fact it allows the store (the best price found) to be moved with the code, letting the purchaser do its job. The new *CODE* rule (that has to instance the rule *CODE* of the code&store style) updates the store of the purchaser, with a new best price, if the price offered by the local space is better than the one in the previous store. Using the **Ambient style** we imagine a mobile "agent" traveling with all its data and exploiting its resources (printer, modem, cellular phone, ...) on different selling-stores looking for the best price for an item. The purchaser could, for example, use

Environment

$$Environment = \{\!|\ (\text{“}s1\text{”} * Shop), (\text{“}s2\text{”} * Shop), (\text{“}agent\text{”} * Ambient)\ |\!\}$$

Shop

$$Shop = \left\{\!\!\left|\begin{array}{l}(\text{“}name\text{”}, k)(\text{“}put\text{”} : PUT), (\text{“}req\text{”}, req),\\ (\text{“}request\text{”} : REQ), (\text{“}get\text{”} : GET)(\text{“}catalog\text{”}, l)\end{array}\right|\!\!\right\}$$

$$PUT = \{\!|\ \uparrow(\text{“}req\text{”}, req), ?(\text{“}name\text{”}, i), (a * Ambient)\ |\!\} \xrightarrow{(z) \leftarrow f(a,i)} \{\!|\ \uparrow(z * Ambient)\ |\!\}$$

where $f(x, y) = (diff(x, y))$

$$REQ = \{\!|\ ?(\text{“}req\text{”}, req)\ |\!\} \longrightarrow \{\!|\ \uparrow(\text{“}req\text{”}, req), (\text{“}wait\text{”})\ |\!\}$$

$$GET = \left\{\!\!\left|\begin{array}{l}\uparrow(a * Ambient), (\text{“}wait\text{”}),\\ ?(\text{“}name\text{”}, z), ?(\text{”}catalog\text{”}, l)\end{array}\right|\!\!\right\} \xrightarrow{(j) \leftarrow f(z,a)} \{\!|\ (j * Ambient)\ |\!\}$$

where $f(x, y) = (concat(x, y))$

Ambient

$$Ambient = \left\{\!\!\left|\begin{array}{l}(\text{“}name\text{”}, k), (\text{“}state\text{”}, s), (\text{‘}bestpricefound\text{‘”}, b),\\ (\text{“}printer\text{”}, p), (\text{“}update\text{”} : UPDATE), (\text{“}print\text{”} : PRINT)\end{array}\right|\!\!\right\}$$

$$UPDATE = \left\{\!\!\left|\begin{array}{l}?(\text{“}bestpricefound\text{”}, b),\\ ?(\text{“}state\text{”}, s),\\ \uparrow(\text{“}catalog\text{”}, l),\\ \textbf{ask}(l < b)\end{array}\right|\!\!\right\} \xrightarrow{(s', b') \leftarrow f(s,l)} \left\{\!\!\left|\begin{array}{l}(\text{“}bestpricefound\text{”}, b'),\\ (\text{“}state\text{”}, s')\end{array}\right|\!\!\right\}$$

where $f(s, l) = (compute(s', l), l))$

$$PRINT = \{\!|\ ?(\text{“}bestpricefound\text{”}, b), \uparrow(\text{“}printer\text{”}, p)\ |\!\} \longrightarrow \{\!|\ (\text{“}outonprinter\text{”}, b)\ |\!\}$$

Table 5.12: Specification of the Purchasing Architecture in the Ambient Style

its printer to print the temporary best price found till that moment.

We give the specification of the architecture of the Purchasing System in the Ambient style in table 5.12. The Shops are Requester and Supplier at the same time. The resources in the Ambient Style shown in Table 5.11 are now portable resources (i.e. a printer, a

scanner, ...). The rules *PUT*, *REQ*, and *GET* do the same operations. The Ambient stores the best price found and contains two rules refining the *CODE* rule in Table 5.11: the rule *PRINT* prints the best price temporary found on the portable printer.

These paradigms could also be composed. For instance in Table 5.11 we allows the agent to read from the shop catalog (i.e to access to some external resources). Therefore, the resulting architecture is a mixture among Code,State&Store style and Ambient style. The *UPDATE* rule updates the best price found if the catalog of the shop offers a better price. These architectures offer different advantages and some of them are better than the other for particular requirements. The designer knows the requirements of the systems that she wants to implement and can choose on the basis of these requirements the most suitable style of mobility. The resulting architecture could also be an integration among different paradigms.

Chapter 6

Summary

Traditional languages, models, and methods used to specify and design applications on a single computer usually lack of abstractions to appreciate and understand the problems raised when the computing platform is a "network computer". Mobility is an obvious example: even if admittedly we could define "mobile code" an application going around on a floppy disk, it is a concept that is especially interesting and complex when a networked programmable infrastructure is available, like an intra-net or even the whole Internet.

In this chapter we have studied how a coordination language can be used to specify and analyze systems including mobile components. The idea consists of having a coordination language that can express a dynamic topology of components and the mobility of code and data.

PoliS and MobiS are not the first formal language used to study systems including mobile entities, as we mentioned in Chapter 1. With PoliS and the model checker, we have built an automatic framework to analyze properties on specifications of mobile code based systems. With MobiS we also offer the ability to model agent mobility as first class in the language so that different mobility paradigms can be specified.

In general, process algebra based languages, like the ones presented in Chapter 1 focus on the notion of *process*, and do not provide the notion of "environment" of the computation. More sophisticated languages offer a concept of environment that we provide with tuple-spaces.

The Chemical Abstract Machine (CHAM) [BB92] has *membranes* that are very similar to our spaces, however the CHAM does not support code mobility as the rules are globally defined outside the "chemical solution" (i.e., the global tuple space).

The ambient calculus [CG98] (briefly described in Chapter 1) provides the notion of *ambient*, which is the mobility unit of the language. *Ambients* are like our spaces, and, like in MobiS, their mobility is first class in the language. The ambient calculus introduces a concept of "step by step" mobility as well. The complexity due to the first class *ambients* mobility make it quite difficult to reason, in particular with tools, on the specifications. We also have to cope with this problem in the case of MobiS, however it is in our future work list the idea of extending the model checker to try to deal with spaces mobility.

Mobile UNITY [MR98] has been used for the specification of physical and logical mobility. It provides a temporal logic that allows reasoning. However no automatic tools exist supporting Mobile UNITY. In [PRM97] Mobile UNITY has been used to formalize some common mobile code paradigms (i.e., Code on Demand, Remote Evaluation, and Mobile Agents). All these paradigms can be also encoded in PoliS and MobiS, and we are looking in the possibility of reason with our automatic tool on them (in the case of PoliS). Furthermore, in Mobile UNITY the dynamic replication of components is not allowed. Therefore, in the Code on Demand paradigm the used code needs to be sent back to the server to be sent again. In PoliS and MobiS the dynamic cloning of code is allowed and the Code on Demand paradigm can be formalized more directly.

Security issues are important in a mobile code setting. Languages such as Klaim [NFP98] and the Ambient calculus [CG98] use "capabilities" on operations, or type systems to face security aspects. Ambient calculus and Seal calculus are based on a "step by step" movement mechanism where components only can move from one domain to another crossing one boundary at a time, instead of on a global location name oriented mobility strategy. Security features are based on this constrained mobility mechanism. Klaim [NFP98] relies on a type system added on top of the model in order to perform static checks on access rights and operations of the system components. In term of language interface improvements we are studying a visual notation for PoliS/MobiS in order to simplify the impact on the users. We are interested in the development of an XML-based abstract syntax [BPSM98b] in order to make PoliS specifications more portable and possibly to be able to be integrated with other XML-based frameworks, like the UML notation [BJR99].

Model checking can be successfully used applied to dynamics of software architectures as proven in [GKC99]. In our approach with PoliS we investigated the use of a model

checker tool for studying mobility aspects of systems. We believe the use of the tool on specifications for mobile code based systems can help in understanding the dynamics of these systems and to avoid mistakes in the design and implementation phase.

In the next parts of this thesis we will take a different perspective on mobile code systems trying to reason on minimal unit of mobility, unit of execution, and basic mobile primitives. This approach will lead us to a prototype system and to the use of XML [BPSM98a] for incremental code mobility.

Part II

Fine-Grained Approach to Mobility: Formalization and Prototyping

Chapter 7

A Fine-Grained Model

The work reported in this part is closely aligned with the investigative style of the formal models community but directed towards identifying opportunities for novel mobility constructs to be used in language design. We are particularly interested in examining the issue of granularity of movement and in studying the consequences of adopting a fine-grained perspective. Simply put, we asked ourselves the question: What is the smallest unit of mobility and to what extent can the constructs commonly encountered in mobile code languages be built from a given set of fine-grained elements? Proper choice of mobility operations, elegant and uniform semantic specification, formal verification capabilities, and expressive power are several issues closely tied into the answer to the basic question we posed.

In the model we explore here the units of mobility are single statements and variable declarations. Location is defined to be a site address and units can move among sites, can be created dynamically, and can be cloned. Complex structures can be constructed by associating multiple units with a process. The process is the unit of execution in our model. In the simplest terms, a process is merely a common name that binds the units together and controls their execution status—more complex structures can be built but they are outside the scope of this paper. All the mobility operations available for units are also applicable to processes. In addition, processes have the means to share code and resources via a referencing mechanism limited strictly to the confines of a single site. A reference can be thought of as a name that allows one process to access some code or data in some other process. References across sites are not permitted but they survive movement, e.g., access is restored when the two processes meet again. As such, unit reference and unit

containment have distinct semantics with respect to both scoping rules and mobility.

Mobile UNITY provides the notational and formal foundations for this study. The new model can be viewed to a large extent as a specialization of Mobile UNITY. This enables us to continue to employ the coordination constructs of Mobile UNITY and its proof logic. The result is a small set of macro definitions that map the fine-grained model proposed here to the standard Mobile UNITY notation, and a semantics specification of the mobility constructs in terms of the coordination language that is at the core of Mobile UNITY.

This application of Mobile UNITY is novel. Mobile UNITY has been used previously in the definition of high level transient interactions (e.g., transiently and transitively shared variables) in both a physical and logical mobile setting [MR98], in formal specification and verification of Mobile IP [MR99], and in the specification and verification of mobile code paradigms (e.g., code on demand, remote evaluation, and mobile agents) [PRM97].

7.1 Model Overview

We now give an informal overview of our model. We consider a network composed of sites. They are the physical locations on which computations take place. Sites may represent physical hosts or separate logical address spaces within a host, e.g., an interpreter. Sites may contain *units* that represent code or data. A code unit need not contain a complete specification of a code fragment, it may even be a single line of code. The variables used in the code units are considered "placeholders" and they do not carry a value (i.e., their value is undefined). Units representing data contain a single variable declaration and they carry the actual value of the variable. The model provides a sharing mechanism between values of variables with the same name in code and data units, thus code can change values of variables in data units during execution.

Because code and data can be split across units, we need to include some notion of composition and scoping. For this purpose we introduce the concept of *process*. Processes are unit containers that reside on the sites. Processes define restricted scopes for the units on the sites. Units can be placed inside a process, i.e., in its "private space". Such units

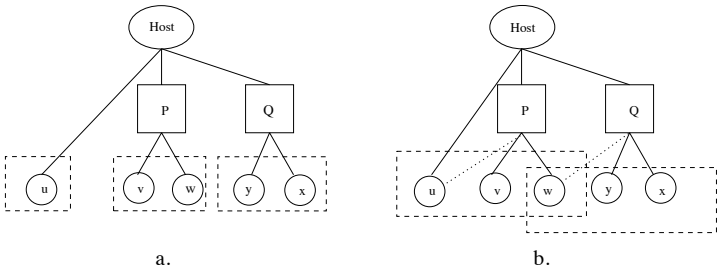

Figure 7.1: Processes, units, and scoping rules.

are said to be *contained* by the process[1]. The scope of a unit contained by a process is the private space of that process, i.e., the space on which the unit is located. The binding mechanisms defined by the model allow sharing among variables with the same name *in the same scope*. The scope of a unit that is not contained in any process (i.e., located directly on the site space) is restricted to the unit itself. In Figure 7.1.a we show an example[2]. The scope of unit v contains also unit w, and vice versa, as they are both contained in process P, while unit u is not contained in any process and its content is not shared with anyone else.

Because it is often necessary to have sharing of units among processes at the same location (e.g., to specify the sharing of a common resource), we allow a process to *reference* a unit contained in another process at the same location. In such a case, the referenced unit is considered to be in the scope of both processes. Processes can also reference units not contained in any process (i.e., located directly in the site). These units can be thought of as library classes or resources provided by the site to all processes located there. Figure 7.1.b shows an evolution of the system from Figure 7.1.a: here unit u is referenced by process P, and units u, v, and w are in the same scope. Unit w is referenced by process Q: since units x, y, and w are in the same scope, sharing applies. Notice that units x and y are not in the scope of unit v.

A process is a unit of execution in the sense that its status constrains the execution of the code belonging to units inside its scope. A process has an activation status that can be manipulated by specific operations. The code units inside the scope of the process

[1]The model presented in this paper is kept simple by not allowing processes to contain other processes. We are investigating this enhancement at the present.

[2]Solid lines represent the containment relation among sites, processes, and units, while dotted lines represent references to units. Dashed rectangles represent a common scope for units.

can only be executed when the process is *active*. Processes constrain the mobility of units as well: the movement of a process implies the movement of all the units contained in it. Referenced units however, are not moved along with the process that refers to them as they are not part of its private space. Furthermore, the binding mechanism inhibits the access to referenced units whenever the referencing process and the referenced unit are not on the same site. It is important to notice, however, that references to units are not discarded at the time of the move; when a referenced unit and the corresponding process become colocated on any site the binding is re-established.

The model also provides mechanisms to generate and duplicate components, to explicitly terminate processes, and to establish or sever a reference between a process and a unit. In the next section we present the structure of the model in some detail.

7.2 Mobile Unity

In this section we provide a more formal treatment of the manner in which the model is built. Along the way, we also describe the Mobile UNITY notation. A Mobile UNITY specification consists of several *programs*, a **Components** section and an **Interactions** section. The program is the basic unit of definition and mobility of the Mobile UNITY system. Figure 7.2 shows a Mobile UNITY solution for the *leader election* problem. N nodes are arranged in a ring each holding a value x. A mobile agent moves around the ring carrying a *token* that it is used to compute the lowest value of the variables x stored on each node. The token value is updated at each node by comparing it with the local value of x. The algorithm is guaranteed to find the leader in exactly one round but for simplicity we allow the agent to circulate indefinitely around the ring. Distribution of components is taken into account through the distinguished location attribute λ associated to each program. Changes in the value of λ denote movement.

The system shown in Figure 7.2 contains two programs, *NodeValue* and *Agent*. The **declare** section of each program contains the declaration of its program variables. The symbol [] acts as a separator. The **initially** section constrains the initial values of the variables. In program *NodeValue* of Figure 7.2, x is initialized using an abstract function id which, given an index i, returns a unique value less that 1000. In the program *Agent* two variables (*token*, and x) are declared. The variable *token* is initialized to 1000 and

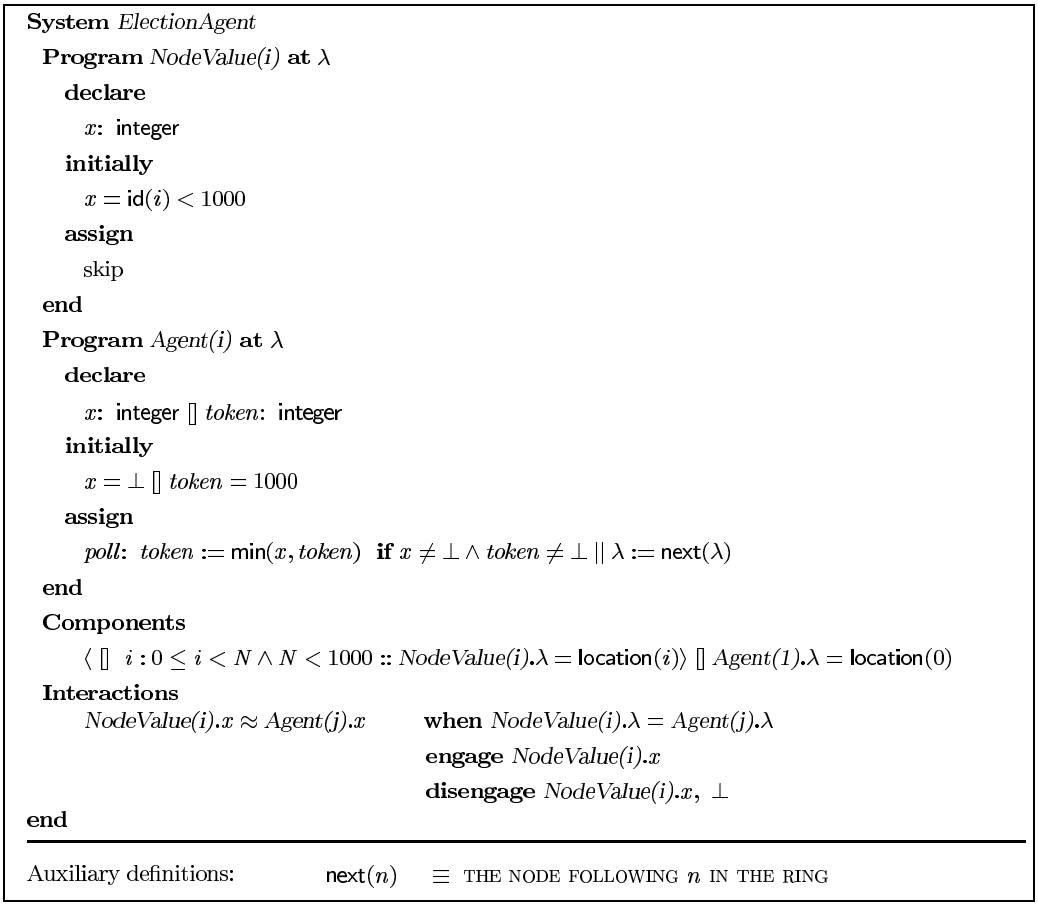

Figure 7.2: A Mobile UNITY system :distributed computation of the minimal value of x.

x is left undefined, i.e., $\perp$. In the **assign** section of program *Agent* the statement named *poll* sets the value of *token* to the minimum between its value and that of x (if x is not $\perp$) and moves the agent to the next node by changing the value of the location attribute λ. The function next returns the next node of the ring, and the symbol || makes the two statements on its left and right to be executed synchronously.

The Mobile UNITY **Components** section defines the components existing throughout the life of the system. Mobile UNITY does not allow dynamic creation of new components. In Mobile UNITY a program definition may contain an index (i.e., i) after the name of the program (i.e., *NodeValue*, or *Agent*). This allows for multiple instances of the same program to be defined in the **Components** section. In Figure 7.2, for instance, N different instances of program *NodeValue* are instantiated and placed at various initial locations

based on their index value[3], initialized using the function location, while only one instance *Agent(1)* of program *Agent* is created.

All the variables of a Mobile UNITY component are considered local to the component. No communication takes place among components in the absence of interaction statements spanning the scope of multiple components. The **Interactions** section contains statements that provide communication and coordination among components. In this example, the **Interaction** section allows the *sharing* of values between the two variables named x in the programs *NodeValue(i)* and *Agent(j)* when they happen to be at the same location: the "." notation is used here to address the variable in the program, and the index i and j are supposed to be universally quantified. Only some of the program instances end up sharing the values of variables x, depending upon their initial location (see function location and subsequent moves). The Mobile UNITY construct $\approx$ defines transient sharing of variables for as long as the **when** condition holds. The **engage** statement defines a common value to be assigned (atomically) to both variables as the **when** condition transitions from false to true. In this example the value assumed by the two variables is the value of the x on the node (*NodeValue(i).x*). It contains the actual value to be used for computing the leader . It is possible to specify also a **disengage** statement that defines the values assumed by the two variables, respectively, the **when** predicate transitions to false. If no **disengage** is specified the variables retain the values they had before the **when** condition became false. In the example, the disengagement value for the x variable on the node (*NodeValue(i).x*) is its current value, while the value of the x carried by the agent (*Agent(j).x*) is set to $\perp$ as it has to carry no value.

The Mobile UNITY execution consists of a fair interleaving of statement executions, including the statements present in the **Interactions** section. The sharing constructs have higher priority and are executed any time a change in the values of the variables involved in sharing happens.[4]

[3]The three-part notation $\langle$**op** *quantified_variables* : *range* :: *expression*$\rangle$ will be used throughout the paper. It is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which **op** is applied.

[4]This is not true for all Mobile UNITY coordination constructs in general but it holds for transient variable sharing, (i.e., for $\approx$).

7.3 Reinterpretation of the Mobile Unity syntax

Mobile UNITY considers a program to be the smallest unit of mobility. In Mobile UNITY every program has a location attribute and the modification of this attribute is denotes the movement of all the code and data in the program. We seek to introduce a finer granularity, one that allows the movement of lines of code or variables as isolated entities. For this purpose we set out to reinterpret the syntax of a standard Mobile UNITY program such that every variable declaration and every labeled statement is interpreted as a stand-alone program, henceforth called a *unit*. A program now becomes only a static unit of definition. Statements and declarations become units of mobility. The Mobile UNITY syntax of a system is preserved. Units generated from the system are formalized using the Mobile UNITY syntax as well. Figure 7.3 shows a possible syntactic transformation for the Mobile UNITY program in Figure 7.2. The small programs shown in Figure 7.3 are, from now on, called *units*.

Some of the variables declared in Figure 7.2 are interpreted as *data units* in Figure 7.3. One can imagine adding a tag **var** in the Mobile UNITY code of Figure 7.2 to specify which variables should be interpreted as data units and which should not. Let us suppose, for instance, that variable x in program *NodeValue(i)* and variable *token* in program *Agent(i)* of Figure 7.2 are treated as data units, i.e., tagged by the keyword **var**. The data unit *p('x,'NodeValue, i)* in Figure 7.3 is generated from the declaration of x in program *NodeValue(i)* (Figure 7.2). The name of all the units is now the constant p, while the three indices after p characterize the unit. Each unit is indexed by its name, the name of the program in which it is defined, and by its instance discriminator. This representation is designed to facilitate the search for units present at some location using the name and/or place of definition. We use a quote to distinguish the actual components from their names, in particular for the first two indices which range over finite enumerations. This notation allows the same names to be present in different program contexts. It is possible, for instance, to define two statements labeled *poll* in two different programs of the same system. The two code units derived will have the same name (i.e., *poll*, the first index), but the second index would be instantiated to different program names. The declaration of x in program *Agent(i)* of Figure 7.2 is assumed not to denote storage, i.e., it is not tagged by **var**. It is only a placeholder needed to accompany the code in the

System *ElectionAgent*
Program *p('x,'NodeValue, i)* **at** λ
declare
x: integer
initially
$x = \mathsf{id}(i) < 1000$
assign
skip
end
Program *p('token,'Agent, i)* **at** λ
declare
token: integer
initially
$token = 1000$
assign
skip
end
Program *p('poll,'Agent, i)* **at** λ
declare
x: integer [] *token*: integer
initially
$x = \perp$ [] $token = \perp$
assign
$token := \mathsf{min}(x, token)$ **if** $x \neq \perp \wedge token \neq \perp$ || $\lambda := \mathsf{next}(\lambda)$ ||
$p('token, \mathsf{find}('token, \lambda) \uparrow 1, \mathsf{find}('token, \lambda) \uparrow 2).\lambda = \mathsf{next}(\lambda)$
end
Components
$\langle$ [] $i : 0 \leq i < N \wedge N < 1000 :: p('x,'NodeValue, i).\lambda = \mathsf{location}(i)\rangle$
[] *p('token,'Agent, 1)*.$\lambda = \mathsf{location}(0)$ [] *p('poll,'Agent,1)*.$\lambda = \mathsf{location}(0)$
Interactions

p('x, i, j).$x \approx$ *p('poll,h, k)*.x	**when** *p('x,i, j)*.λ = *p('poll,h, k)*.λ
	engage *p('x,i, j)*.x
	disengage *p('x,i, j)*.x, $\perp$
p('token,i, j).*token* $\approx$ *p('poll,h, k)*.*token*	**when** *p('token,i, j)*.λ = *p('poll,h, k)*.λ
	engage *p('token,i, j)*.*token*
	disengage *p('token,i, j)*.*token*, $\perp$

end

Auxiliary definitions: $\mathsf{next}(n)$ $\equiv$ THE NODE FOLLOWING n IN THE RING

Figure 7.3: Fine-grained restructuring of the *ElectionAgent* System.

statement *poll*. Therefore, the declaration of the x in *Agent(i)* is not translated as a data unit in Figure 7.3. The computation in *poll* will actually use the data in the x variable located on each node. This is made possible by the sharing mechanism defined in the **Interactions** section. Notice that a unit capturing a variable declaration also contains

the corresponding initialization statement for the declared variable. The assign section of a data unit does not contain any statement as the unit only declares a variable.

In order to overcome the difficulty of dynamically creating components in Mobile UNITY we assume to have a sufficiently large number of instances of components initially located in a sort of "ether". We formalize this by saying that they reside at an undefined location $\lambda = \epsilon$. In this manner, whenever we need to duplicate or instantiate a new component, we simply change the location of some component in the ether from undefined to an actual location.

The *code unit p('poll,'Agent, i)* of Figure 7.3 is generated from the statement *poll* in Figure 7.2. The first index of the code unit is the label of the statement. The second index is instantiated, like in data units, i.e., the name of the program the unit comes from, and the third is the index that allows multiple instances of the same unit. The code of the *poll* statement is part of the **assign** section. All the variables used in the statement are declared (in the **declare** section) and initialized as unbound, i.e., $\perp$. This initialization underlines the fact that this unit only contains code and that the variables are merely placeholders (i.e., they do not contain real values until placed in a context that provides sharing with data units).

As we want the *token* unit and the *poll* unit to move together, like in the example in Figure 7.2 where they move within the program context, we now have to modify the *poll* code by adding an explicit command for the movement of the unit *token* as well. The function find returns, given the first index of a unit, the last two indices for a unit present at a given location (the notations $\uparrow 1$ and $\uparrow 2$ are used to address the first and second field of the returned value of find). The search can be done in two ways, by name or by name and place of definition: looking for the last two indices (i.e., the name of the program the unit is derived from and the index for multiple instances), or only on the last index giving the name of the program containing the unit. The example shows a generic *find* that returns both last indices (as the system contains only one definition of the data unit *token*). The semantics of *find* will be defined in Section 7.5. The statement in unit *poll* uses the value of the x present at each site to perform its computation. The sharing mechanism in the **Interactions** section allows the sharing of the value of x (and of *token*) between the data unit carrying the value and the placeholder variable in the code unit *p('poll,'Agent,i)*. The **Components** section in Figure 7.3 places the units in the same

location as those of Figure 7.2.

The re-interpretation of the Mobile UNITY system as a fine-grained mobile system allows units to move as separate entities and code and data to be stored in different components. In the resulting model we lose the notion of scoping previously associated with the individual programs. It may appear that since, data and code are separated, their simultaneous movement and value sharing among variables have to be programmed explicitly, in the code and in the **Interactions** section. These difficulties can be avoided, however, if we introduce the notion of a container which can be constructed dynamically, can move its entire contents of data and code units as a whole, and provides for automatic sharing of like-named variables appearing in data and code units placed inside the container. We will refer to this kind of container as a *process* because we intend to use such components not only as dynamically structured programs but also as basic units of execution. As a matter of fact, as shown in the next section, code units will be prevented from executing whenever they reside outside the confines of a process. A process is seen as a program and therefore formalized as *p(name,prog,i)*, where the first index is the name of the process, the second is the name of the program from which the initial contained units are defined into, and the third index is the one allowing multiple instances.

The three indices defining a process are also used as a location name and used in the definition of location for units inside the processes: while the location attribute of a process is always set to a name of a host (as processes reside directly on the hosts), units location attributes can be strings composed of the concatenation of the name of the host they reside on and of the three process indices they are in (if they are in a process). In this model processes cannot contain other processes while in Chapter 8 we show an enhancement of the model that allows process and unit locations to be complex concatenation of strings (a host and several process indices tuples).

With the introduction of processes we are now able to move lines of code, single variables, or complex groups of units and not only programs like in the example in Figure 7.2. The notion of scope introduced by processes also helps in the simplification of the sharing mechanisms between variables. Variables with the same name in the scope of the same process may be considered as sharing their values. The explicit sharing mechanism available in Mobile UNITY can then be avoided by exploiting scoping. By providing a standard set of sharing rules designers do not need to touch the **Interactions** section. For instance,

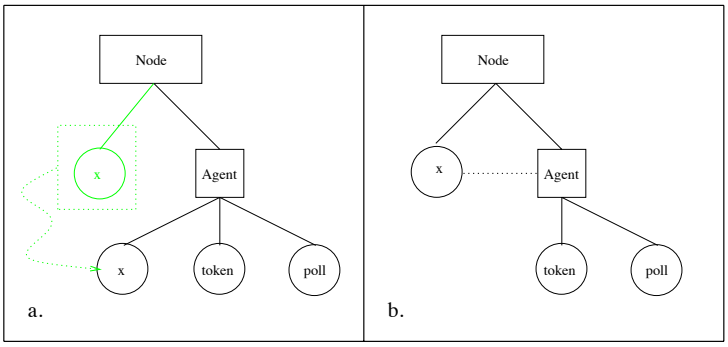

Figure 7.4: A node in the leader election solution with processes.

the sharing among the *token* variable in the data unit *token* and the same variable in the code unit *poll* had to appear explicitly in Figure 7.3. Further refinement of our new model will allows binding by name to be established automatically between two variables within the same scope.

The explicit sharing between the variable x on the node and its counterpart in the poll code can also be eliminated if somehow we are able to pull within the scope of *poll* the variable x residing on the same node. This can be accomplished in two ways. First, we can explicitly move the variable x (on the node) within the scope of the process P (Figure 7.4.a) embodying the agent. Second, we can add a new construct called external reference (or simply reference), which extends the scope of the agent process to include x without moving it (Figure 7.4.b). In the next section we introduce the use of these constructs for the specification of the leader election problem.

7.4 Mobility Constructs

The previous section hinted at the key points of departure from Mobile UNITY, and at the manner in which we will ultimately reduce a notation for fine-grained code mobility back to the essence of Mobile UNITY. Central to our model is the interplay among the notions of execution, scoping, containment, and location. Mobility not only determines the set of resources that are available at a given location, but also allows the dynamic reconfiguration of the code and data associated with a given process. In this section we describe in more detail the set of constructs available in our model. In the next section, we will use Mobile UNITY to provide formal semantics to these constructs.

In Section 7.3 we have shown a mobile agent solution for the leader election problem. We used the example to explain the re-interpretation of the Mobile UNITY system in terms of units. In this section we refine the solution to the leader election problem given in Section 7.3 exploiting more fully than before the features and the constructs of our model: code, data and agents mobility, as well as built-in scoping rules. The solution assumes that no nodes are initially able to take part in a leader election. The distributed algorithm is started by injecting into the ring a process that contains the necessary knowledge about the distributed computation—a *voter*. This process clones itself repeatedly until the whole ring is populated with voters. Interestingly, voters do not contain the logic associated with the token, i.e., they do not know how to compare the node's value with the token's value—the *poll* strategy. The knowledge about this key aspect of the algorithm is injected into the ring in a separate step of the computation in the form of a code unit which is placed on an arbitrary node of the ring. Each voter is able to detect the presence of the *poll* code unit on its node and move it into its own scope, thus effectively enabling the execution of the unit. The poll code unit has access to a node-level data unit that contains the node value. This enables the comparison needed to vote. Again, a self replicating scheme is employed, where each voter passes on a copy of the unit to the next node in the ring. This structure of the system, where the poll strategy is kept separate and is loaded dynamically into the voter, enables the dynamic reconfiguration of the ring. This happens when a new code unit that contains a different poll strategy is injected in the ring. Again, voters detect its presence on their sites and replace the old strategy with the new one. Finally, when the token is injected into the ring the actual leader election starts.

Our example, despite its simplicity, highlights many of the *leitmotifs* of mobile code: simultaneous migration of the code and state associated with a unit of execution, dynamic linking (and upgrade) of code, and location-dependent resource sharing. For instance, our solution can be easily adapted to an active network scenario where a new service (in our case the ability to perform leader election) is deployed in the network, and some of its constituents (in our case the poll strategy) are dynamically upgraded over time.

A formal specification of our leader election algorithm is shown in Figure 7.5, while Figure 7.6 shows its graphical representation. The specification uses the fine-grained mobile code constructs of our model.

Figure 7.5: Leader Election in Mobile UNITY extended with fine-grained mobililty constructs.

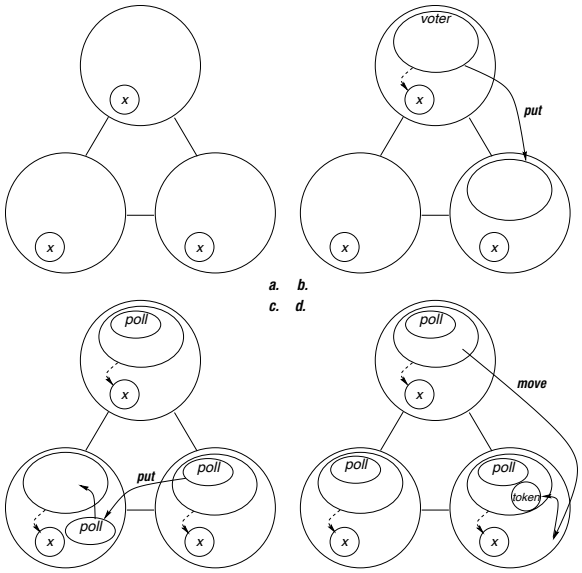

Figure 7.6: Leader election with mobile code.

The upper part of the specification contains three program definitions. The programs in Figure 7.6 are not indexed with an i unlike in Figure 7.2 as with the introduction of processes programs are only units of definition and they are not instantiated.

NodeDefinition specifies a single data unit x associated with a node. The type declaration for this integer variable is prepended by the keyword **var** which characterizes the variable as a data unit. The initialization of the variable is not defined here: the predicate **newData**, used in the **Components** section will provide the initial value to be assigned when the unit is instantiated. In this way we allow different instances of the same unit to be initialized with different values. The first parameter of **newData** is the name of the variable, the second is the name of the program in which the unit is defined, the third is the location where the unit has to be placed, and the forth is the initial value. The program *TokenDefinition* specifies a data unit associated with the variable *token*. The values of these two variables are accessed (through sharing) by code units specified by the program *PollActions*. The latter contains a single statement *poll*, which describes the polling strategy. As discussed in the next section, the formal semantics of the model prescribes the execution of this statement to be prevented when the corresponding code unit is not within the scope of any process. Thus, the comparison in *poll* is performed only when the corresponding code unit is co-located in a voter process that also contains the

data unit corresponding to *token*. In this case, the binding rules of the model, expressed using the transient variable sharing abstraction provided by Mobile UNITY, effectively force the same value in both *token* variables, hence enabling the comparison specified by *poll*. Simultaneously, an additional auxiliary boolean variable *voted* is set to signal to the enclosing voter, again by means of sharing of the variable *voted*, that the token needs to be passed along the ring.

Voters are specified by the program *VoterActions*, that declares the variables mentioned so far and an additional boolean *startup* that is used to determine whether it is necessary to perform some initialization tasks, i.e., cloning the voter itself on the next node to perform the initial deployment of processes in the ring, and acquiring a reference to the node's value. These tasks are performed simultaneously by the statement *startVoter*, which also resets *startup* to prevent the creation of multiple clones of the voter.

In *startVoter*, cloning is performed by the **put** operation. It executes only if the voter that is invoking the operation does not immediately precede in the ring node(0) where the whole computation started. This guarantees that each node hosts a single voter. The statement uses some of the auxiliary definitions shown at the bottom of the figure. In particular, here and thisNode are just renamings of the location variable λ in the voter and of the head function that operate on it, respectively. They serve the sole purpose of improving readability. While the location of a process is always set to the name of a site (as processes reside directly on the site), unit location can refer to sites or to processes. In the latter case, the location is defined as the concatenation of the name of the site the unit reside on and of the name of the process that holds it. This is useful in invoking the **put** operation whose most general form is $\mathbf{put}(name, prog, id, location_{dest})$ where the first three parameters are the three indices of the component to be copied and $location_{dest}$ is a location that represents the destination of the copy. Another form, $\mathbf{put}(name, location_{cur}, location_{dest})$, is also provided. It is actually used in the example to "query" the scope defined by $location_{cur}$ for the second and third indices of the component given the *name* (i.e., first index).

As will become clear in the next section, copying takes place behind the scenes by picking a fresh component from the ether and setting its location to the one passed as a parameter. Like most of the operations provided in our model, the **put** operation is defined on components, i.e., both on processes and units. Hence, in the case of processes

the copying is performed recursively on the process and on all its constituent units. In the case of **put**, the bindings that a process may have established are not preserved as a consequence of this copy operation, i.e., all the variables are restored to their initial values. This represents a "weak" form of copying. Our model provides also a stronger notion with the **clone** operation, which preserves all the bindings owned by the process.

The statement *startVoter* establishes also a reference to the variable x, whose value is contained in a data unit instantiated on each site. To understand in more detail this latter aspect, let us take a brief detour and jump temporarily to the **Components** section, to look at the initial configuration of the system. The first statement uses the macro **newData** to indicate the creation of a data unit named x using the definition provided in the program *NodeDefinition*, assigns to it the value i, and places it on the i^{th} node. Since the statement is quantified over the number N of nodes in the system, each node hosts an instance of this data unit as a result of the operation.

Similarly, the other three statements in the **Components** section create on the first node respectively the data unit for the token, the code unit for the poll strategy, and the voter process. Given the nature of our model, which enables movement to the level of a single Mobile UNITY variable or statement, it is interesting to note how *VoterActions* actually represents the unit of definition for a number of units, namely, the data units corresponding to *voted* and *startup*, and the code units corresponding to *startVoter*, *linkCode*, and *passToken*. In principle, each of these could be moved or copied independently. Since this is not the case in this example, they have been grouped together under *VoterActions*. This simplifies the text of the specification by minimizing the number of **Program** declarations, and also enables the creation of a single process that automatically contains instances for all the aforementioned units by using **newProcess**. Finally, note how the value of a process is its activation status, i.e., either ACTIVE or INACTIVE.

Now, let us return to the **reference** operation in *startVoter*. Thanks to the binding rules, this operation establishes a transient sharing between the variable in the data unit x defined in *NodeDefinition* and the declaration in the voter. Note how, similarly to what was described for **put**, only the name of the data unit x is specified, while its indices is determined by implicitly querying the node. The model provides also the inverse operation **unreference**.

The statement *linkCode* takes care of replicating the poll strategy and, possibly, of

substituting the new poll code for the old one. It executes only when the exists function in the guard evaluates to true. The function exists, formally introduced in the next section, effectively models the aforementioned query mechanism, and enables *linkCode* to execute only when a code unit with name *poll* is found on the node. If the unit is found, the **move** operation brings it within the process, thus enabling its execution. Simultaneously, a copy of the unit is sent to the next node in the ring via a **put**, provided that the next node is not node(0). At the same time, if a pre-existing *poll* unit is found in the process the **destroy** operation removes it from the system.

Finally, *passToken* handles the movement of the token. Again, the query mechanism is used to get implicitly the identifier of any *token* data unit present on the node and **move** it within the process to establish the proper bindings. After the poll is performed, i.e., *voted* is set to true, the token is moved from the scope of the voter to the next node in the ring.

7.5 Formal Semantics

Our general strategy is to reduce the new model for code mobility to a specialization of the standard Mobile UNITY notation and proof logic. The first step, explained in the previous sections, shows how we reinterpret a notation which looks very close, if not identical, to that of Mobile UNITY by simply treating each variable declaration and statement as a separate, independent program. Multiple instantiations of each such fine-grained program, called a unit, are defined in the **Components** section. Once this transformation from a concrete to an abstract syntax is completed, the parts of the model still missing are the mechanics of data sharing within the confines of each process, the control over the scheduling of statements for execution, and the definition of the various mobility constructs. Our strategy is to capture all these semantic elements as statements present in the **Interactions** section of the Mobile UNITY system and to disallow the designer from adding anything else to the **Interactions** section. The result is a specialization of Mobile UNITY to the problem of fine-grained mobility. The fact that the entire semantic specification can be reduced to a small set of coordination statements attests to the flexibility of Mobile UNITY. In the remainder of the section we consider in turn the topics of scoping, statement scheduling, mobility constructs, and creation predicates. From now on we use

$$
\begin{array}{rl}
\mathsf{find}(u,l) \equiv & \langle \mathsf{min}\ i,j : u_{i,j}.\lambda = l :: (u,i,j)\rangle \\
\mathsf{find}(u,i,l) \equiv & \langle \mathsf{min}\ j : u_{i,j}.\lambda = l :: (u,i,j)\rangle \\
\mathsf{exists}(u,l) \equiv & \langle \exists i,j :: u_{i,j}.\lambda = l\rangle \\
\mathsf{exists}(u,i,l) \equiv & \langle \exists j :: u_{i,j}.\lambda = l\rangle
\end{array}
$$

Figure 7.7: Specification of the functions find and exists.

the compact notation $c_{i,j}$ to mean $p(c,i,j)$, i.e., the instance j of the component named c extracted from program i. Throughout this section we also assume that:

- Each component, (i.e., data unit, code unit, or process) $c_{i,j}$ is characterized by its location ($c_{i,j}.\lambda$), request field ($c_{i,j}.\rho$) designed to hold mobility commands the system is expected to execute on its behalf, and type ($c_{i,j}.\tau \in \{\text{DATAUNIT}, \text{CODEUNIT}, \text{PROCESS}\}$).

- Each process $q_{i,j}$ is also characterized by an implicitly specified set of contained units (those located within the process), a set of referenced units ($q_{i,j}.\gamma$), and its activation status ($q_{i,j}.\omega \in \{\text{ACTIVE}, \text{INACTIVE}, \text{TERMINATED}\}$).

The designer does not need to refer to any of these attributes even though they are essential to the formal semantic definition.

When writing code, the designer will typically refer to a component's name (e.g., c) rather than its fully qualified name (e.g., $c_{i,j}$) consisting of the three indices (i.e., c, i, j) defining the component name, program, and index, respectively. Given the name, the other identifiers can be extracted easily by employing the functions find and exists defined in Figure 7.7.

The find function finds an instance of the component named u on the location l. The name of the program the unit is derived from (i.e., i) can be added as a parameter in order to constrain the search only to units derived from a particular program definition; the same is true for the function exists. Processes, like other units, also have three indices: the first index is the name of the process, the second is the name of the program the units in the process are derived from (e.g., the process *voter* created with **newProcess** in the **Components** section of Figure 7.5), and the third is the instance discriminator.

7.5.1 Scoping Rules

Since a code unit can only access its own variables, the mechanism by which we establish scoping and access rules is that of forcing variables with the same name and present in the same scope (i.e., contained in the same process) to be shared. This can be readily captured by employing one of the high level constructs of Mobile UNITY, transient variable sharing across programs ($A.a \approx B.b$ **when** p). The predicate p controlling the sharing simply needs to capture the scoping rules. Figure 7.8 shows how these rules can be stated as two Mobile UNITY coordination statements. Statement 7.1 handles sharing between a variable in a data unit and a variable in a code unit, while statement 7.2 defines the sharing between two variables in data units.

Statement 7.1 states that variables[5] $u_{i,h}.x$ and $w_{j,k}.x$ share the same value when $u_{i,h}$ is a data unit and $w_{j,k}$ is a code unit, and the two units are within the same process, or either the data unit or the code unit is referenced by the process owning the other unit and the two units are on the same site. The **engage** value is the value of the variable in the data unit. The two **disengage** values are the actual value shared for the data unit variable, and the undefined value for the code unit variable, respectively—variables in code units are not supposed to carry a value unless they are sharing it with a data unit. The function sharing tells if two units have a common "parent" (a parent can be the process within which they are located or the one which references them), i.e., the units are in the same scope. In turn, sharing uses the functions childOf($v_{j,k}, u_{i,h}$), that determines whether $v_{j,k}$ is child of $u_{i,h}$ (i.e., $v_{j,k}$ is a unit contained in $u_{i,h}$), and referencedBy($v_{j,k}, u_{i,h}$), that determines whether $v_{j,k}$ is referenced by $u_{i,h}$.

Statement 7.2 defines sharing between variables in two data units. The variables must have the same name in the same scope. Sharing takes place under the same conditions of statement 7.1, except that both variables are in data units. The **engage** clause forces the two variables to share the maximum value. Different policies can implement a different semantics for reconciliation of values. As no **disengage** is specified the variables retain the values they had before the **when** condition became false. The update of all shared

[5]The formulae in Figure 7.8 and following assume that variable sharing is well-defined, i.e., it takes places only among variables which actually appear in the specification of a unit according to the program definition. Also, distinguished variables like λ and τ are never shared. The formal definition of these conditions is omitted for the sake of brevity.

$$
\begin{array}{rll}
u_{i,h}.x \approx w_{j,k}.x & \textbf{when } u_{i,h}.\tau = \text{DATAUNIT} \wedge w_{j,k}.\tau = \text{CODEUNIT} \ \wedge & \\
& (u_{i,h}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(u_{i,h}.\lambda) \ \vee & \\
& (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) & (7.1) \\
& \textbf{engage } u_{i,h}.x & \\
& \textbf{disengage } u_{i,h}.x, \ \bot & \\
u_{i,h}.x \approx w_{j,k}.x & \textbf{when } u_{i,h}.\tau = w_{j,k}.\tau = \text{DATAUNIT} \ \wedge & \\
& (u_{i,j}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(w_{j,k}.\lambda) \ \vee & \\
& (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) & (7.2) \\
& \textbf{engage } \mathsf{max}(u_{i,h}.x, w_{j,k}.x) & \\
\textbf{inhibit } u_{i,h}.s & \textbf{when } u_{i,h}.\tau = \text{CODEUNIT} \ \wedge & \\
& (\langle \forall p, m, n : p_{m,n}.\tau = \text{PROCESS} \wedge (\mathsf{childOf}(u_{i,h}, p_{m,n}) \ \vee & \\
& \mathsf{referencedBy}(u_{i,h}, p_{m,n})) :: p_{m,n}.\omega \neq \text{ACTIVE} \rangle \vee & (7.3) \\
& \langle \exists x :: u_{i,h}.x = \bot \rangle) &
\end{array}
$$

Auxiliary definitions:

$$
\begin{array}{rl}
\mathsf{sharing}(u_{i,h}, w_{j,k}) = & \langle \exists p, m, n :: (\mathsf{childOf}(w_{j,k}, p_{m,n}) \wedge \mathsf{referencedBy}(u_{i,h}, p_{m,n})) \vee \\
& \quad (\mathsf{childOf}(u_{i,h}, p_{m,n}) \wedge \mathsf{referencedBy}(w_{j,k}, p_{m,n})) \rangle \\
\mathsf{childOf}(v_{j,k}, u_{i,h}) = & \begin{cases} \mathit{true} & \text{if } v_{j,k}.\lambda = u_{i,h}.\lambda \circ (u, i, h) \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathsf{referencedBy}(v_{j,k}, u_{i,h}) = & \begin{cases} \mathit{true} & \text{if } (v, j, k) \in u_{i,h}.\gamma \\ \mathit{false} & \text{otherwise} \end{cases}
\end{array}
$$

Figure 7.8: Bindings among units using variable sharing and statement inhibition.

variables must happen in the same atomic step as the assignment to any of them. However, sharing is specified separately from the (possibly many) assignments that may change the value of a variable. To accomplish this, Mobile UNITY has a two-phased operational model where the first phase involves an ordinary assignment statement execution and the second is responsible for propagating changes to shared variables. We call the statements that execute in the second phase *reactive statements*. Logically, the set of reactive statements are executed to fixed point right after each non-reactive statement and one reactive statement may trigger the execution of other reactive statements. Transient sharing is ultimately defined using reactive statements [MR98], but this is outside the scope of this paper.

7.5.2 Statement Scheduling

In Mobile UNITY, each statement is assumed to be executed infinitely often in an infinite execution, i.e., weakly fair selection of statements is the basis for the scheduling process. The coordination constructs of Mobile UNITY include a construct for guard strengthening called **inhibit**. In **inhibit** s **when** p, for instance, the statement s continues to be selected as before, but its effect is that of a skip whenever the condition p is not met. We take advantage of this construct in statement 7.3 of Figure 7.8 to inhibit statements not in the scope of an active process, and statements that have unbound variables. A variable appearing in a statement is always *unbound* if it is not shared with a variable present in a data unit.

7.5.3 Mobility Constructs

The designer views the **move** construct as a mechanism by which a component at one location is relocated to another. The new location may be a known site or a known process. This form of the **move** construct:

$$\textbf{move}(compName, currentLocation, newLocation)$$

is actually a special instance of the more general form in which the identity of the unit is already known. One can simply determine the identity by employing the function find as in[6]

$$\textbf{move}(\mathsf{find}(compName, currentLocation), newLocation).$$

If multiple instances of the same unit exist one is selected[7]. In order to explore the manner in which we assigned semantics to the mobility constructs associated with our model we will focus our presentation on the general form of the construct. Moreover, we will assume that the unit in question is a process named q with identifier (i, j) destined for location l:

$$\textbf{move}(q, i, j, l).$$

[6]Throughout, we assume that $\textbf{move}((q, i, j), l)$ is unambiguously reducible to $\textbf{move}(q, i, j, l)$.

[7]We chose to pick up the instance with minimum index.

Our general strategy is to treat the operation as a macro reducible to a simple local assignment statement to the distinguished variable ρ (see Figure 7.9):

$$\rho := (\text{REQ}, \text{MOVE}, (q, i, j, (j, l))$$

where the first two fields of the record stored in ρ indicate the propagation status (i.e., an initial request) and the nature of the request (i.e., a **move**).

We delegate the actual execution of the operation to a series of coordination statements built into the **Interactions** section. The coordination statements propagate the request to the contained units and ultimately carry out the migration of the individual components to the new location. All these actions are executed atomically because they are encoded as reactive statements that execute to fixed point before the system is allowed to take any other action. The first thing that happens is to have the request transferred in the form of a command to the process q. The result is that $q_{i,j}.\rho$ is assigned the request with a propagation status of EXEC:

$$q_{i,j}.\rho := (\text{EXEC}, \text{MOVE}, (q, i, (j, l))$$

while the attribute ρ of the unit issuing the request is cleared. Of course, in general it might be the case that a unit requests its own movement and one needs to distinguish between the two cases as made evident in Figure 7.10.

If, for the sake of simplicity, we assume that the only units contained by q are $d_{m,h}$ and $s_{k,n}$, the next reaction being triggered leads to having the process ready to start the move, a fact indicated by dropping the propagation status

$$q_{i,j}.\rho := (\text{MOVE}, (j, l))$$

while simultaneously propagating the command to the contained units (see Figure 7.10), e.g.,

$$\begin{aligned} d_{m,h}.\rho := &\quad (\text{EXEC}, \text{MOVE}, d, m, h, (h, l \circ (q, i, j))) \\ s_{k,n}.\rho := &\quad (\text{EXEC}, \text{MOVE}, s, k, n, (n, l \circ (q, i, j))) \end{aligned}$$

Figure 7.10 defines the function $\mathcal{F}$ that computes, in a command-specific manner, the arguments needed by the contained units. In this case, the location to where they

$$
\begin{array}{rl}
\textbf{move}(u,i,j,l) & \equiv \rho := (\text{REQ}, \text{MOVE}, u, i, j, (j, l)) \\
\textbf{put}(u,i,j,k,l) & \equiv \rho := (\text{REQ}, \text{PUT}, u, i, j, (\mathsf{getid}(u,i), l)) \;||\; k := \mathsf{getid}(u,i) \\
\textbf{clone}(u,i,j,k,l) & \equiv \rho := (\text{REQ}, \text{CLONE}, u, i, j, (\mathsf{getid}(u,i), l)) \;||\; k := \mathsf{getid}(u,i) \\
\textbf{destroy}(u,i,j) & \equiv \rho := (\text{REQ}, \text{DESTROY}, u, i, j, ()) \\
\textbf{activate}(u,i,j) & \equiv \rho := (\text{REQ}, \text{ACTIVATE}, u, i, j, ()) \\
\textbf{deactivate}(u,i,j) & \equiv \rho := (\text{REQ}, \text{DEACTIVATE}, u, i, j, ()) \\
\textbf{terminate}(u,i,j) & \equiv \rho := (\text{REQ}, \text{TERMINATE}, u, i, j, ()) \\
\textbf{new}(u,j,k,l) & \equiv \rho := (\text{REQ}, \text{NEW}, u, j, \mathsf{getid}(u,j), (l)) \;||\; k := \mathsf{getid}(u,j) \\
\textbf{reference}(u,i,j,v,k,h) & \equiv \rho := (\text{REQ}, \text{REFERENCE}, u, i, j, (v,k,h)) \\
\textbf{unreference}(u,i,j,v,k,h) & \equiv \rho := (\text{REQ}, \text{UNREFERENCE}, u, i, j, (v,k,h))
\end{array}
$$

Auxiliary definitions:

$$
\begin{array}{l}
\mathsf{getid}(name) \equiv \mathsf{get2nd3rd}(\mathsf{find}(name, \epsilon)) \\
\mathsf{getid}(name, i) \equiv \mathsf{get3rd}(\mathsf{find}(name, i, \epsilon))
\end{array}
$$

Figure 7.9: Mapping mobility constructs to Mobile UNITY statements.

need to move is the relocated process. Since further propagation is no longer possible the commands drop the propagation status in the next step

$$
\begin{array}{rl}
d_{m,h}.\rho := & (\text{MOVE}, (h, l \circ (q,i,j))) \\
s_{k,n}.\rho := & (\text{MOVE}, (n, l \circ (q,i,j)))
\end{array}
$$

The last step is the change in location of each of the units (Figure 7.11). Given the semantics of Mobile UNITY, this may happen in any order but the reactive statements will be executed again and again until fixed point is reached, i.e.,

$$q_{i,j}.\lambda = l \;\wedge\; d_{m,h}.\lambda = l \circ (q,i,j) \;\wedge\; s_{k,n}.\lambda = l \circ (q,i,j)$$

If an attempt is made to move a unit before the containing process, an apparently inconsistent state is reached in which the unit is located inside of a nonexistent process but this is corrected as soon as the process move is complete. Thus the command completes always in a consistent state.

All other constructs function in a similar manner except that not all the commands are propagated to the contained units. For instance, **terminate** affects only the status of the process. The function toPropagate used in Figure 7.10 is designed to control the propagation process: the propagating constructs are **move**, **put**, **clone**, and **destroy**. The construct getid returns the three-part identity of a component located in the ether. A minimal lexicographical value for the triplet is selected. The two functions get2nd3rd

$$w_{j,k}.\rho = \perp \ \textbf{if}\ w_{j,k} \neq u_{i,h} \ ||\ u_{i,h}.\rho = (\text{EXEC}, command, u, i, args)$$
$$\textbf{reacts-to}\ w_{j,k}.\rho = (\text{REQ}, command, u, i, h, args) \tag{7.4}$$

$$u_{i,h}.\rho = (command, args)\ ||\ \langle||\ v, n, m : \mathsf{childOf}(v_{n,m}, u_{i,h}) \wedge \mathsf{toPropagate}(command) ::$$
$$v_{n,m}.\rho = (\text{EXEC}, command, v, n, \mathcal{F}(command, u, i, v, n, m, args)) \tag{7.5}$$
$$\textbf{reacts-to}\ u_{i,h}.\rho = (\text{EXEC}, command, u, i, args)\rangle$$

Return values for $\mathcal{F}$:

$$\mathcal{F}(\text{MOVE}, u, i, v, n, m, (h, l)) = (m, l \circ (u, i, h))$$
$$\mathcal{F}(\text{PUT}, u, i, v, n, m, (k, l)) = (\mathsf{getid}(v, n), l \circ (u, i, k))$$
$$\mathcal{F}(\text{CLONE}, u, i, v, n, m, (k, l)) = (\mathsf{getid}(v, n), l \circ (u, i, k))$$
$$\mathcal{F}(\text{DESTROY}, u, i, v, n, m, ()) = ()$$

Figure 7.10: Modeling the actions of the run-time support.

and get3rd return the second and third indices, and the third index, respectively, given the indices triple returned by the function find. The complete list of commands and the corresponding formalization appear in Figures 7.9 and 7.11.

7.5.4 Creation Predicates

The three macros **newData**, **newCode**, **newProcess** are used in Figure 7.5 for the instantiation of new components. **newData** is defined in two forms, the first allows the setting of the initial value as a parameter (i.e., v)[8]. The second uses the initial value defined in the program. The constructs used in Figure 7.5 are special instances of more general form: for instance, **newData**(u, n, l, v) is a special form of **newData**$(u, n, getid(u, n), l, v)$. The function getid (shown in Figure 7.9) has two parameters in this case as we know one of the indices (i.e., the program name n). Table 7.12 contains the semantics of these predicates. **newData** states that a new data unit is located at location l and that setting the initial value for its variable is v. **newCode** states that a new code unit is located at location l. The predicate **newProcess** locates a process at location l, with status s. The predicates **newData** and **newCode** are used to define the initial location of all the units that have

[8]The **newData** predicate is used in the **Components** section in order to define the instantiation of new data units. The implicit quantification over the variables used in the **Components** section is generally restricted to some proper range. In case of variable names the range is set to the names appearing in the unit (i.e., the case of x.

$$u_{i,h}.\lambda := l \;\; \textbf{if } (u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon \;||$$
$$u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{MOVE}, (h, l)) \tag{7.6}$$

$$u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \;\; \textbf{if } (u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \;||$$
$$u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{PUT}, (k, l)) \tag{7.7}$$

$$u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \;\; \textbf{if } (u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \;||$$
$$u_{i,h}.\rho := \bot \;||\; \langle \forall x :: u_{i,k}.x := u_{i,h}.x \rangle \textbf{ reacts-to } u_{i,h}.\rho = (\text{CLONE}, (k, l)) \tag{7.8}$$

$$u_{i,h}.\lambda := \bot \;\; \textbf{if } u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{DESTROY}, ()) \tag{7.9}$$

$$u_{i,h}.\omega := \text{ACTIVE} \;\; \textbf{if } u_{i,h}.\omega = \text{INACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho = \bot$$
$$\textbf{reacts-to } u_{i,h}.\rho = (\text{ACTIVATE}, ()) \tag{7.10}$$

$$u_{i,h}.\omega := \text{INACTIVE} \;\; \textbf{if } u_{i,h}.\omega = \text{ACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot$$
$$\textbf{reacts-to } u_{i,h}.\rho = (\text{DEACTIVATE}, ()) \tag{7.11}$$

$$u_{i,h}.\omega := \text{TERMINATED} \;\; \textbf{if } u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||$$
$$u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{TERMINATE}, ()) \tag{7.12}$$

$$u_{i,h}.\lambda := l \;\; \textbf{if } u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l) \;||\; u_{i,h}.\rho := \bot$$
$$\textbf{reacts-to } u_{i,h}.\rho = (\text{NEW}, l) \tag{7.13}$$

$$u_{i,h}.\gamma := u_{i,h}.\gamma \cup \{(v, j, k)\} \;\; \textbf{if } v_{j,k}.\tau \neq \text{PROCESS} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \wedge$$
$$v_{j,k}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho = \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{REFERENCE}, (v, j, k)) \tag{7.14}$$

$$u_{i,h}.\gamma := u_{i,h}.\gamma \setminus \{(v, j, k)\} \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{UNREFERENCE}, (v, j, k)) \tag{7.15}$$

Figure 7.11: Migrating components.

$$\begin{aligned}
&\textbf{newData}(u, n, k, l, v) && \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.x = v \\
&\textbf{newData}(u, n, k, l) && \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.x = \mathsf{initial}(n, x) \\
&\textbf{newCode}(u, n, k, l) && \equiv u_{n,k}.\lambda = l \\
&\textbf{newProcess}(u, n, k, l, s) && \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.\omega = s \wedge \\
& && \langle \forall u' : \mathsf{datadefined}(u', n) :: \textbf{newData}(u', n, \mathsf{getid}(u', n), l \circ (u, n, k)) \rangle \wedge \\
& && \langle \forall u' : \mathsf{codedefined}(u', n) :: \textbf{newCode}(u', n, \mathsf{getid}(u', n), l \circ (u, n, k)) \rangle
\end{aligned}$$

Figure 7.12: Constructs for the instantiation of components.

to be inside the process.

Chapter 8

An Enhanced Fine-Grained Model

Mobile code technologies presented in Chapter 1 usually implements a model where unit of mobility coincide with the unit of execution. In the most of the cases the model is flat like the one presented in Chapter 7. A flat model does not allow the unit of execution to contain other units of execution. In this chapter we want to loose this constrain extending the model in Chapter 7.

In this chapter we present an extension of the fine-grained model presented in Chapter 7. In this enhanced model processes can contain not only units but also other processes, therefore generating a hierarchy of processes on the hosts. Hierarchical scoping is combined with dynamic reconfiguration of process structures. This enhanced model is a natural extension of the previous model.

The notation of the location that in the previous chapter was associated only with hosts is now associated also with processes that become real locations for other units and processes. As the modifications to the model only affect the notion of location, the real changes to the semantics are minimal. The enhanced model allows the specification of a powerful system with a location based hierarchical structure that still does not have a corresponding developed technologies (only a similar attempt has been developed in Telescript [Whi96]). We now give details of the enhancement.

8.1 The Enhancement with Scoping

A process is now a container not only for units but also for other processes. The general structure of a host is now a tree. Figure 8.1.a shows the representation of a host containing

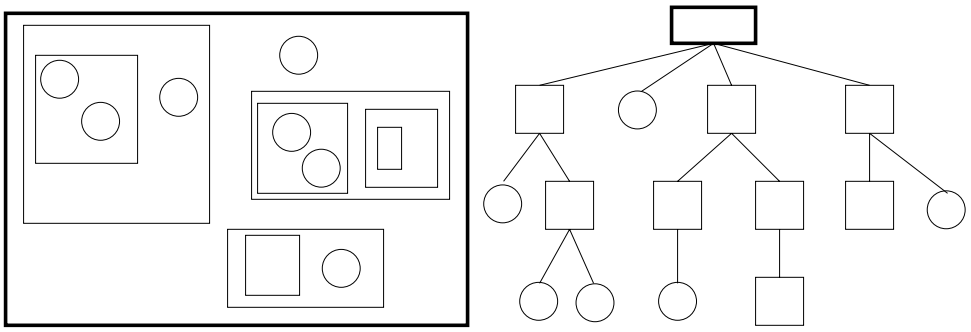

Figure 8.1: The structure of a host in the enhanced model (a) and the corresponding tree topology (b).

nested processes and Figure 8.1.b shows the corresponding tree topology [1]. Nested structures promise added flexibility in the organization of resources and systems. Nevertheless, they are not typically encountered in mobile code languages. The extension turns out to be a natural step, merely a simple refinement of the notion of location. A location is no longer the concatenation of at most two names (i.e., the host name and, if needed, the process name), but an arbitrarily long concatenation of names, reflecting process nesting.

Every nested process acts as a block structured context: all the data and code in the block are considered *local* to the process and cannot be accessed from the upper blocks. However, the inner blocks can access the content of the outer blocks. The binding mechanism presented in Table 7.8 is readily extended to accommodate the new hierarchical process structure. Figure 8.2 shows an example of binding in the tree. Variables with the same name in the scope of the same process (like x in the units v and u) are bound and share the same value (as in the "flat" model). As one might expect, structural changes due to mobility of code fragments leads to corresponding changes in scope and data access. Let us consider again Figure 8.2. Notice that the data unit w contains a declaration for x. While the data unit v is present, the variable x in code unit u is bound to the declaration of x in v. If unit v moves away, the x in code unit v becomes bound to x in w. In general, the binding mechanism binds a variable in a code unit to the "closest" declaration for that variable found in the path to the root (i.e. the host) of the tree.

The access to referenced units must also be adjusted for use with the hierarchical model.

[1] The rectangular thicker boxes represent hosts, normal boxes are processes, and circles are units.

We constrain the scope of the referenced unit only to the peer units in the referencing process; however it is still possible to explicitly formalize the access to a referenced unit also for the lower level units in the branch. Let us consider, for instance, Figure 8.3.a: the variable x in the code unit v is bound to the declaration of x in data unit z (we assume that the unit z and process P are on the same host). The declaration of x in z, however, cannot be bound to the x in code unit u as this is at a lower level (i.e., it is not a peer unit). In some cases the designer may want to let the x in v to be bound to the x in z: to do this she can either put another reference to x from process Q or exploit the reference from process P and introduce a new data unit w declaring x at the peer level of z in process P. In this case the binding mechanism allows the sharing between the two declarations of x in data units in the same scope (i.e., w and z), and w is the closest data unit declaring x with respect to unit u, then the binding is established between the two x (see Figure 8.3.b).

In the hierarchical perspective a new operation can be added to the model to be able to constrain the access of a unit only to peer units. At the moment the access strategy is a hierarchical access, where lower level units can be bound to an upper level unit (in case it contains the closest declaration for a certain variable). The new operation could constrain the accessibility of a unit only to peer units. In order to do so units should have an attribute indicating their access type: the automatic sharing mechanism looking for the closest instance of a variable would now have to consider this attribute in computing it. For instance, let us consider again Figure 8.3.b in this context: if the unit w had access right set to PeerAccess, unit u would never be able to share the value of its variable, as it is not a peer unit.

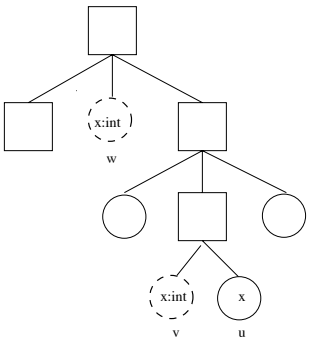

Figure 8.2: Scoping in the enhanced model: the dashed circles represent data units.

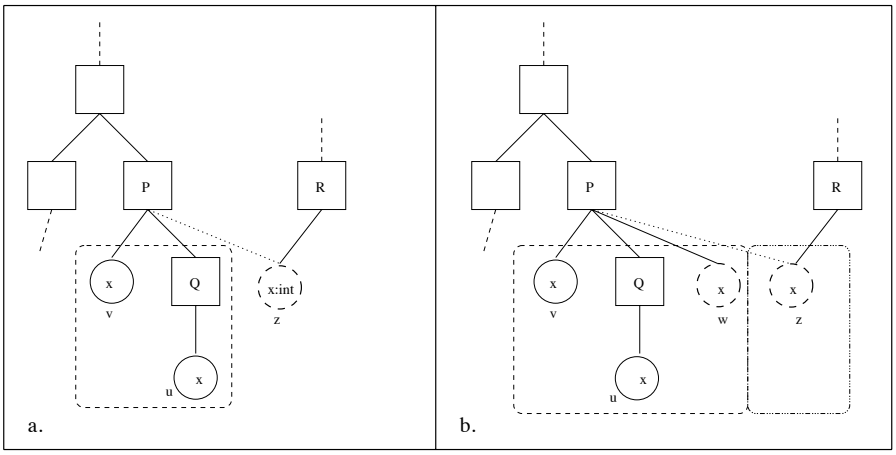

Figure 8.3: Referencing in the hierarchical model

The hierarchical model requires little changes in the semantics of the constructs defined in Table 7.11. As the model in general is enhanced only modifying the notion of location, the changes required in the semantics are minimal. The operations **move**, **put**, **clone**, can now place a process putting it inside another process, not only on hosts. Let us consider, for instance, the **move** operation (7.6) in Table 7.11. The check

$u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)$

ensures that a process is only be moved on a host, and not inside another process (i.e., l is a host location). In the modified **move** for the hierarchical model this check disappears and the new formalization of **move** is:

$u_{i,h}.\lambda := l$ **if** $(u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon) \mid\mid u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{MOVE}, (l))$

Figure 8.4 contains the updated table for all the constructs, refining Figure 7.11.

Moreover, the operations **activate**, **deactivate**, and **terminate** have to be propagated to all the child processes of the input process. For this reason the function $\mathcal{F}$ has to be defined also for these constructs (it simply returns the () value). Figure 8.5 shows the new functions.

$$u_{i,h}.\lambda := l \;\; \textbf{if } u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{MOVE}, (h, l)) \tag{8.1}$$

$$u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \;\; \textbf{if } u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{PUT}, (k, l)) \tag{8.2}$$

$$u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \;\; \textbf{if } u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \;||\; \langle \forall x :: u_{i,k}.x := u_{i,h}.x \rangle \textbf{ reacts-to } u_{i,h}.\rho = (\text{CLONE}, (k, l)) \tag{8.3}$$

$$u_{i,h}.\lambda := \bot \;\; \textbf{if } u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{DESTROY}, ()) \tag{8.4}$$

$$u_{i,h}.\omega := \text{ACTIVE} \;\; \textbf{if } u_{i,h}.\omega = \text{INACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho = \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{ACTIVATE}, ()) \tag{8.5}$$

$$u_{i,h}.\omega := \text{INACTIVE} \;\; \textbf{if } u_{i,h}.\omega = \text{ACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{DEACTIVATE}, ()) \tag{8.6}$$

$$u_{i,h}.\omega := \text{TERMINATED} \;\; \textbf{if } u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{TERMINATE}, ()) \tag{8.7}$$

$$u_{i,h}.\lambda := l \;\; \textbf{if } u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l) \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{NEW}, l) \tag{8.8}$$

$$u_{i,h}.\gamma := u_{i,h}.\gamma \cup \{(v, j, k)\} \;\; \textbf{if } v_{j,k}.\tau \neq \text{PROCESS} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \wedge v_{j,k}.\lambda \neq \epsilon \;||\; u_{i,h}.\rho = \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{REFERENCE}, (v, j, k)) \tag{8.9}$$

$$u_{i,h}.\gamma := u_{i,h}.\gamma \setminus \{(v, j, k)\} \;||\; u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{UNREFERENCE}, (v, j, k)) \tag{8.10}$$

Figure 8.4: Migrating components: enhanced model.

$$\begin{aligned}
\mathcal{F}(\text{MOVE}, u, i, v, n, m, (h, l)) &= (m, l \circ (u, i, h)) \\
\mathcal{F}(\text{PUT}, u, i, v, n, m, (k, l)) &= (\mathsf{getid}(v, n), l \circ (u, i, k)) \\
\mathcal{F}(\text{CLONE}, u, i, v, n, m, (k, l)) &= (\mathsf{getid}(v, n), l \circ (u, i, k)) \\
\mathcal{F}(\text{DESTROY}, u, i, v, n, m, ()) &= ()
\end{aligned}$$

Figure 8.5: Updated functions return values.

Chapter 9

Lilliput: a Fine-Grain Mobility Prototype

The fine-grained model presented in Chapter 7 describe a different approach to mobility from a formal point of view. In this chapter we show a prototype of the approach that wants to show the implementability of the idea. We describe the design of the prototype, the name of which is LILLIPUT. The Java API (Application Program Interface) is then contained in Appendix A.

The LILLIPUT implementation follows the formal specification given in Chapter 7. The input language of the system is a simplified Mobile UNITY [MR98] (where no interaction section can be defined, the symbol **var** is added in the **declare** section, and all the variable have integer type). The complete grammar of the input language can be found in Appendix A. A compiler then translates the input document in a Java document that is going to be compiled and executed on the Java Virtual Machine. The pseudo-Mobile UNITY specification defines the system initial configuration distributed over a network of hosts and the specification of the variables and code of the system. The compiler translates every statement into a code unit and every variable declaration with its initialization in a data unit (it generates Java classes for them). The **Components** section is translated in a main method calling some other methods to send units and processes to different locations. Figure 9.1 depicts the process of translation of the input document into Java documents.

Once the system is initialized, units and processes are located on different hosts. An engine is started on each host to maintain consistent sharing among variables (according to rules specified in Chapter 7), inhibiting code from execution, and executing the mobility constructs invoked by the code units. When the system is running units and processes

move around and each engine on hosts has the responsibility to keep the binding in a consistent state.

We will go through the details of the system next.

9.1 The Lilliput System

We now introduce the details of LILLIPUT starting from the interface with the programmer. The language used to prepare input documents for LILLIPUT is based on the Mobile UNITY notation [MR98] that will be translated into a set of Java files (Figure 9.1). A small example will help in the description from now on. Figure 9.2 contains the specification of a simple input system that shows how LILLIPUT can migrate processes and units, update code when needed (through injection of new code in existing processes, and disposal of code out of date), bind variables and execute code units. The example allows a variable (i.e, x) with state, to be moved from a process to another on a different host. Once the variable reaches its destination, code for executing an increment of the variable may execute. At a later time, the code for the increment needs to be updated with a more "efficient" increment strategy. Therefore, new code is shipped and injected into the remote process while disposing of the old code.

The system shown (i.e., **System** *Example*) consists of three programs. The first program (i.e., **Program** *Migration*) contains a declaration (with keyword **declare**) of variables x, and *flag*, their initialization (with keyword **initially**) and an assignment section (with keyword **assign**) containing two statements labelled with *migrate* and *ship*, respectively. The statement *migrate* specifies the movement of the variable x from localhost to process Q on $HOST2$ and the assignment of the variable *flag* to *true*. The statement *ship* allows the movement of a statement labelled with *increment* to the same process Q

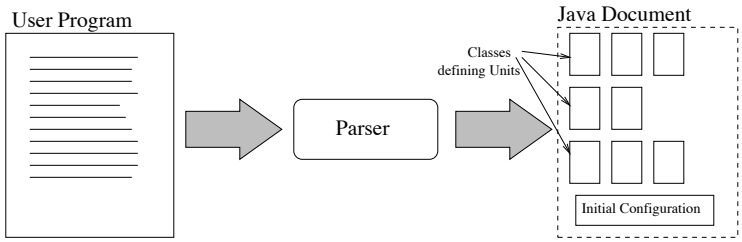

Figure 9.1: Translation of the Input Document.

```
System Example
  Program Migration
    declare
      x: var integer
      flag: var boolean
    initially
      x = 0
      flag = false
    assign
      migrate:   move(x, localhost, find(Q, HOST2)) || flag := true
      ship:      move(increment, localhost, find(Q, HOST2))  if flag = true
  end
  Program Increment
    declare
      x: integer
    assign
      increment:   x := x + 1  if ¬find(increment, Update, here)
      change:      destroy(increment, Increment)  if find(increment, Update, here)

  end
  Program Update
    declare
      x: integer
    assign
      increment:   x := 2x

  end
  Components
      newProcess(P, Migration, HOST1, ACTIVE)
      newProcess(Q, Increment, HOST2, ACTIVE)
      newCode(increment, Update, HOST1)
end
```

Figure 9.2: An example of system as input in LILLIPUT.

on *HOST*2. The second program (i.e., *Increment*) increments a variable x, and contains another statement to update the code *increment* once new code for "more efficient" increment computation has been received. The third program (i.e., *Update*) contains the increment code that is going to be used for the update defined above.

In order to understand the meaning of these programs we need to give details on how this specification is interpreted by LILLIPUT. Programs are considered "units of definition" of the data and code of the system. Every variable declared with the keyword **var** is interpreted as a data unit of the system and a unit is created, using the initial value assigned in the **initially** section. In Figure 9.3 we show the data unit, implemented in

```
import lilliput.*;

public class DUxMigration extends LilliDU{
  public DUxMigration(){
    super("x","Migration",0);
  }
}
```

Figure 9.3: The Java representation of the data unit for variable x.

```
import lilliput.*;

public class CUmigrateMigration extends LilliCU {
  public CUmigrateMigration(){
    super("migrate","Migration");
    vars.add(new Variable("flag"));
  }
  public void perform(){
    LilliEngine e=LilliEngine.getEngine();
    LilliDU d=(LilliDU)e.find("x");
    e.move(d, "HOST"","Q");
    LilliDU d'=(LilliDU)vars.search("flag");
    d'.value=TRUE;
  }
}
```

Figure 9.4: The Java representation of the code unit for statement migrate.

Java, that corresponds to the declaration of variable x of program *Migrate* in Figure 9.2. The Java class extends a data unit class (i.e., `lilliDU` from package `lilliput`), and calls the constructor with the name of the unit (i.e., `x`), the program it comes from (i.e., `Migration`), and the initialization value (i.e., `0`) as parameters[1]. A similar data unit is defined for the variable *flag*.

Every statement of the program (i.e., every labelled line after the **assign** keyword) is interpreted as a code unit. In Figure 9.4 we show the LILLIPUT code unit corresponding to the statement labelled with *migrate* in program *Migration* of Figure 9.2[2]. Notice that

[1]From now on we refer to a data unit with the name of the variable that it represents. This can lead to ambiguities in case two variables with same name are member of two different units, however this is not going to be the case in our example.

[2]From now on we refer to a code unit with the label of the statement that it represents. This can lead to ambiguities in the case of statement *increment*, defined both in program *Increment* and *Update*. We

all the variables in the code units are appended to a list of variable names (i.e., `vars`). No value for variables is recorded in code units as variables are only placeholders to be bound to actual data units variables. The `perform()` method contains the actual code of the unit. First a reference to the local LILLIPUT engine needs to be determined (i.e., `e`). Then, a search by name for the unit is performed on localhost (with `find()`). At this point the movement primitive, `move()`, can be invoked to move the unit `d` to the new host and then to the scope of process `Q`. The statement that in Figure 9.2 is specified after the symbol || is executed in a sequential manner in the code unit[3]. The variable `flag` is updated. Notice that the `search()` method used to find the `flag` variable from the list of variables bound to the code unit, has a different meaning from the method `find()`, used to find the data unit to be moved. While `search()` looks only in the list of variables bound to the code, `find()` searches for a unit on which a mobility primitive has to be applied (`find()`, in fact, may also be used for searching code units and processes that need to be migrated or replicated).

The declaration of variable x in program *Increment* and in program *Update* are not preceded by the keyword **var**, and therefore no data units for them need to be created. This is justified by the fact that x here is only a dummy variable for the definition of the statement *increment* in the **assign** section. The *increment* statement in program *Update*, the *change* statement in program *Increment*, and the *ship* statements in program *Migrate* are interpreted as code units similar to the one in Figure 9.4.

The **Components** section contains the initialization setting for the system. In Figure 9.2 two processes, P and Q are created and placed on location $HOST1$ and $HOST2$, in an active state, respectively, using the construct **newProcess**. The second parameter of **newProcess** is the name of the program from which units being part of the scope of the process must be taken from. In case of process P, for instance, the units x, *flag*, *migrate*, and *ship* are put in its scope. In case of process Q only the unit *increment* is placed in the scope. The code unit *increment* of program *Update* is placed in the host "library" of $HOST1$ for future use using the command **newCode** [4]. In Figure 9.5 we show the corre-

will distinguish in this case specifying also the name of the program.

[3]Semantically, this interpretation of concurrency as interleaved actions is valid: the code unit is executed in an atomic fashion by LILLIPUT of the host.

[4]The construct **newData** is provided as well in order to allow the creation and location of data units(as defined in the semantics in Chapter 7.

```
import lilliput.*;

public class Lilliput implements LilliConstants{
  public static void main(String[] args){
    LilliEngine lilli= new LilliEngine();
    LilliProcess P= new LilliProcess("P","Migration");
    LilliProcess Q= new LilliProcess("Q","Increment");
    DUxMigration duXM= new DUxMigration();
    DUflagMigration duFlagM= new DUflagMigration();
    CUmigrateMigration cuMigrateM= new CUmigrateMigration();
    CUshipMigration cuShipM= new CUshipMigration();
    CUincrementIncrement cuIncrementI= new CUincrementIncrement();
    CUchangeIncrement cuChangeI= new CUchangeIncrement();
    CUincrementUpdate cuIncrementU= new CUincrementUpdate();
    lilli.addInScope(P,duXM);
    lilli.addInScope(P,duFlagM);
    lilli.addInScope(P,cuMigrateM);
    lilli.addInScope(P,cuShipM);
    lilli.addInScope(Q,cuIncrementI);
    lilli.addInScope(Q,cuChangeI);
    lilli.addInScope(Q,cuIncrementU);
    lilli.newProcess(P,"Migration","HOST1",ACTIVE);
    lilli.newProcess(Q,"Increment","HOST2",ACTIVE);
    lilli.newCode(cuIncrementU, "HOST1");
  }
}
```

Figure 9.5: The Java representation of the **Components** section.

sponding initialization class (i.e., `Lilliput`) generated from the **Components** section of Figure 9.2. The `LilliEngine` object is created. Two processes and all the units to place in their scope are created and the scoping relationships are established with `addInScope()`. Then, the `newProcess()` method is called to place the processes in the right status and on the right location.

We suppose that a LILLIPUT engine is started on every host of the system. Figure 9.6 shows the architecture of a host in the system. Once the units and the processes are placed on the right initial locations, the system begins its routine operations. The listener thread waits for incoming entities arriving on the host and puts them in a queue. In the meanwhile, a spawn *interpreter* thread gets entities from the queue, and handles their relocation on the host. The interpreter also takes care of bindings among the variables of

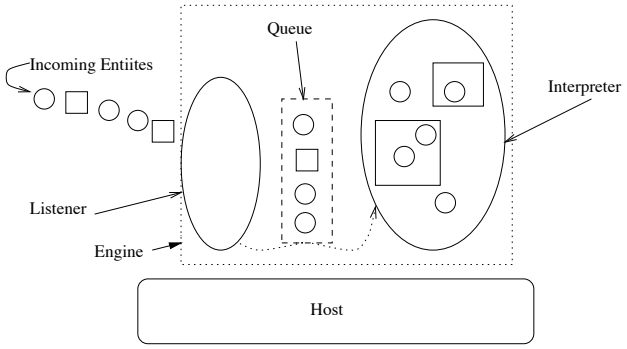

Figure 9.6: A host of the LILLIPUT system.

all the units (in the same scope), of enabling of code units for execution (once all their variables are bound and the process they are in active), of the (non deterministic) choice of an enabled code unit for execution, and of its execution.

Each code unit can execute simple operations, like the increment of a variable (i.e. as the *increment* statement in Figure 9.2), or mobility operations like the `move()` in the code unit `CUmigrateM`. Mobility operations are implemented in LILLIPUT, and they consist of constructs of migration, creation, replication, and referencing reflecting the model described in Chapter 7.

Going back to our example, consider the initial situation also depicted in Figure 9.7, where the process P is on $HOST1$ and Q on $HOST2$. The interpreter on P will be able to bind the variable *flag* in data unit `duFlagM` with the variable *flag* in code unit `cuMigrateM` (both in the scope of process P). This will allow the code unit `cuMigrateM` to be enabled for execution (as P, defining the scope for that unit, is active). The code unit `cuIncrementI` is also executable, however its guard is always false (until the `cuMigrateM` sets it to `TRUE` while executing). The interpreter on $HOST1$ chooses non deterministically code units for execution among the enabled ones. The interpreter will eventually pick up for execution the `cuMigrateM` unit. The execution of the unit performs a migration (with `move()`) of the sibling unit `duXM`, if found (otherwise an exception is risen), that will be transferred to $HOST2$ and sets the *flag* to *true*.

On $HOST2$, the listener puts the data unit just received in a queue and the local LILLIPUT interpreter will place the unit in the scope of process Q, where the unit is destined. The interpreter on $HOST2$ was idle as no units were enabled for execution: the code unit `cuIncrementI` had unbound variables (i.e., the variable x) as no data unit was

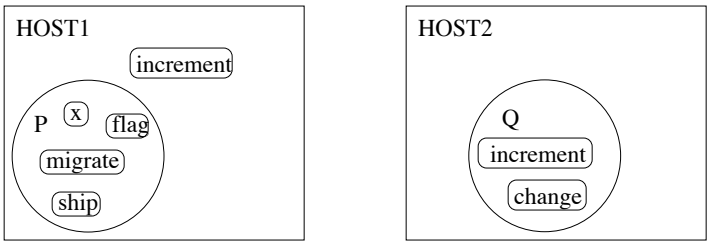

Figure 9.7: The initial system configuration.

present in the process's scope. When the data unit `duXM` arrives the interpreter is able to bind the two x of the data and code unit and the code unit becomes ready for execution. The execution simply performs increments on the value of the variable x in the bound data unit `duXM`, changing the actual value in the data unit. When the code unit `cushipM` is eventually executed on $HOST1$ (as the flag now is true) the code unit `cuIncrementU` is shipped to the process Q on $HOST2$, thus enabling the code unit `cuchangeI` that will dispose of the old code for the increment so that the new code may be used. In order to simplify the example we have used a flag to determine when it was time to ship the new code for the computation of the increment. However, it is possible to refine this example defining more sophisticated policies for deciding when it is necessary to ship new code (i.e., after a request, or after checking some performance parameters). The potential of LILLIPUT are discussed further in Section 9.4.

In the next section we describe the architecture of the LILLIPUT implementation in details.

9.2 The Architecture of the System

LILLIPUT has been implemented in Java 1.2 in about 1200 lines of code. The communication of the hosts involved in the system is handled using the μ-CODE toolkit [Pic98]. μ-CODE is a light weight mobile code system with a small set of abstractions to provide the shipping and the fetching of code and objects across a network of hosts. The integration of μ-CODE with LILLIPUT can be clarified considering again Figure 9.6. The LILLIPUT engine on a host spawns a μ-CODE thread (i.e., the listener) to listen for incoming elements. The LILLIPUT method for the migration of elements outside the host is `move()`. `move()` is implemented exploiting μ-CODE methods for migrating units/processes to other hosts.

```
public class LilliEngine implements LilliConstants{
  public void move(LilliElement e, String location, String ProcessName)
  public LilliElelement cloning(LilliElement e)
  public LilliElelement put(LilliElement e)
  public LilliElement neww(Class c)
  public void destroy(LilliElement e)
  public void activate(LilliProcess p)
  public void deactivate(LilliProcess p)
  public void terminate(LilliProcess p)
  public void reference(LilliProcess p, LilliUnit u)
  public void unreference (LilliProcess p, LilliUnit u)
  public LilliElement find(String s)
}
```

Figure 9.8: The LILLIPUT engine interface.

In case the element to be transferred is a unit, `move()` instructs μ-CODE to migrate the object instanced for the unit and the class describing the unit (i.e., a class like the ones in Figure 9.3 and in Figure 9.4). Notice that in case of a code unit the object carries no status for the variables used in the code (as they are only placeholders to be bound on destination). In case the element to be moved is a complete process, the movement of all the units in the scope of the process needs to be triggered. The objects of the process and all the units are moved through μ-CODE together with all the classes for all the units. In any case all the binding between data unit variables and code unit variables need to be severed before migration. This is particularly important in case of referencing to units in libraries of the host that are not moved together with the referencing processes. The engine interface is shown in Figure 9.8.

The engine provides the mobility methods available to the programmer. The `move()` method has already been described. The two methods `cloning()` and `put()` allow replication of processes and units, with status or without (i.e., with or without initial setting), respectively. The `neww()` method creates a new instance of a unit or process, given the class. This is used to dynamically instantiate classes creating new units or to generate new processes. `reference()` and `unreference()` establish and severe a reference between a unit and a process. `activate()`, `deactivate()`, and `terminate()` change the status of a process to `ACTIVE`, `DEACTIVATE`, or `TERMINATED`, respectively. The `destroy()` method explicitly eliminates the process or unit.

```
public void run(){
  while(true){
    engage();
    eval();
  }
}
private void engage(){
  merge();
  bind();
  enable();
}
private void eval(){
  LilliCU c=null;
  c=pickUp();
  execute(c);
}
```

Figure 9.9: The LILLIPUT interpreter thread main cycle.

The LILLIPUT interpreter main cycle is shown in Figure 9.9. The interpreter first phase is implemented by the `engage()` method. In this phase the interpreter gets (with `merge()`) the first of the received elements from the queue (Figure 9.6). Then, it puts it in the right place on the host, either linked to a process or on the host library. The `bind()` method binds the variables of the code units to the data units (in the same scope) so to enable code units for execution. The `enable()` method then searches for the code units ready for execution (i.e. the ones in active processes). The second phase of the interpreter consists in the execution of a code unit. The `eval()` method uses `pickup()` to choose one enabled unit for execution in a non deterministic fashion. Then the `execute()` method calls the `perform()` method of the chosen code unit to actually execute the unit code. The code may, like in case of the code unit in Figure 9.4, contain migration operations (i.e., move). If this is the case, μ-CODE is called to handle the migration process.

9.3 Implementation Details

We now show, using UML class diagrams [BJR99], the main components of the LILLIPUT system architecture (Figure 9.10).

Every entity derives from the abstract class `Element`. An `Element` has a name, and

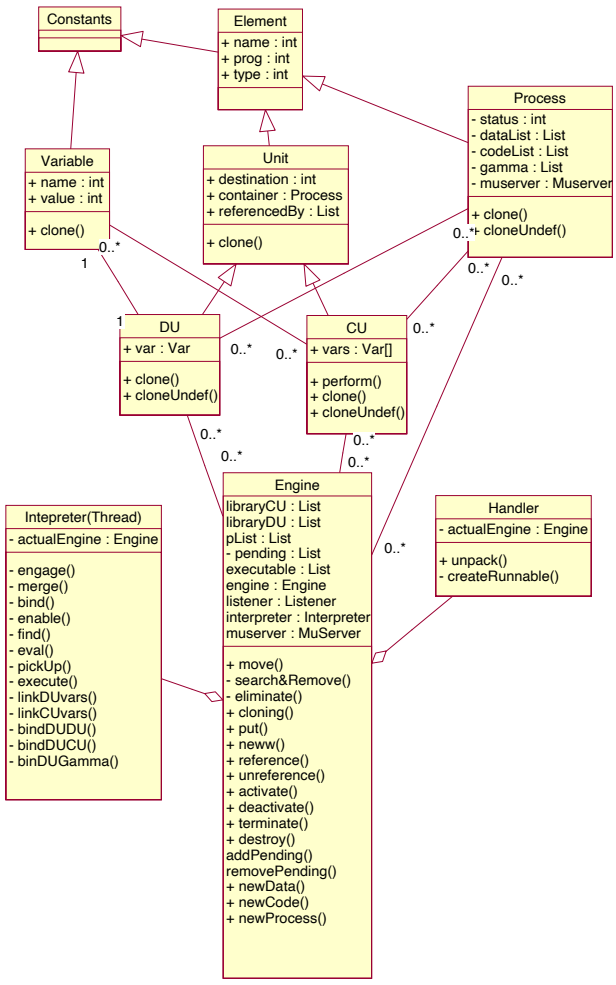

Figure 9.10: The Class Diagram of LILLIPUT.

a program name (i.e., the name of the program it is derived from). The element also has a type that will be used to distinguish data, and code units and processes.

The class `Constants` defines the constants of the system, while the class `Variable` defines a prototype for a variable, with a name and a value. The class `Unit` and `Process` inherit from the `Element` class. A `Unit` class carries the attribute `container` pointing at the containing process for the unit. The attribute `destination` is set in the movement phase when the unit needs to migrate inside a remote process, it carries the name of the process. The `referencedBy` list records the processes referencing the unit. Referencing is the ability to access the contents of a unit without being in the same process context (see Chapter 7 for details). The `clone()` method clones the unit. The `Process` class contains

the attribute `status` , i.e., the activation status of the process. The lists `dataList` and `codeList` define references to the contained data and code units, respectively. The `gamma` list contains links to the referenced units. The `muServer` is the local muserver to store the classes of the units contained. The method `clone()` clones the process. `cloneUndef()` performs a clone with initialization values to variables in the units set. The classes `DU` and `CU` define the abstract data and code units. The inherit from `Unit` and they are refined in real data and code unit in the Java document output of the front-end of LILLIPUT (like in Section 9.1).

The `DU` class defines an abstract data unit. It contains a variable declaration. Every data unit derived from the program defined by the programmer by compilation is an extension of the `DU` abstract class.

The `CU` class defines an abstract code unit. It contains the code to be executed (single statements). It stores a list of variables, that are the variables used in the code statement. The method `perform` is implemented by the real code units and will contain the code to be executed. For simplicity the code can contain arithmetic expressions and mobility constructs only. The `perform` method executes the code of the unit. If the code contains mobility constructs they are invoked on the public mobility methods of the engine on the host (see Section 9.3.1), containing static methods.

Both code and data unit classes contains a method `clone` that clones the unit and a method `cloneUndef` to clone and set initial values. This is used in order to implement the **put/clone** primitives defined in the model (Chapter 7).

We now describe the main engine of the system and its classes.

9.3.1 The Engine

The engine is the main element located on each host. The engine class contains structures used for the evolution of the system. In particular the engine contains:

- the processes list (i.e. the list of processes on the host): `plist` ;
- the lists of "pending for entering" entities : `pending` ;
- code units, and data units in the "library" of the host: `dataList` and `codeList` ;
- the list of executable code units: `executable` .

- the μ-Code server `muserver` used to listen to the incoming elements (and to implement the `move` operations).

The engine also implements the methods defining the mobility primitives. These methods are called in the `perform()` method of the code units.

- `move()` : to move the entity (unit or process). Implemented on top of μ-Code. This is the only operation involving remote hosts. As we suppose the other operations only act locally; the method generates a μ-Code object (i.e., a group) that will be sent the engine on the destination site. The destination engine listens for incoming groups through a μ-Code server.
- `put()` and `clone()` : to duplicate entities;
- `new()` : to create an instance of a class or process on the location.
- `activate` / `deactivate` / `terminate` : to change the status of processes;
- `destroy` : to destroy an entity;
- `reference` / `unreference` to reference/unreference a unit: the operation can only act locally on the host. I.e., if I want to unreference a unit but it is not here the operation throws an exception.
- `newData()` : this operation allows the setting to a location of a new data unit;
- `newCode()` : this operation put a code unit to a location;
- `newProcess` : it puts a new process to a location.

There are also some auxiliary methods: `eliminate()` , `searchAndRemove()` , `addPending()` , and `removePending()` that we do not describe for brevity.

9.3.2 The Interpreter

The interpreter waits for the engine to put elements in the `pending` queue, then gets the elements, puts them in the right lists of the engine, binds the variables, enables the code units for execution and execute one of them.

We now describe each method in details:

- `engage()` . `engage()` calls two other methods in order to get incoming elements, merge them to the right lists, and bind the variables. `engage` uses the following methods:
  - `merge()` : this method executes a `synchronized` method `removePending()` of the engine that acts on the `pending` queue. Whenever the listener puts something in the queue with the engine method `addPending()` the interpreter is woken-up. Then, the elements then are put in the right lists on the engines.
  - `bind()` : the method is called to stabilize the system after the arrival of new elements. It binds the variables with the same name contained in two data units in the same scope and variables with same name in data and code units in the same scope.
  - `enable()` : it checks the code units on the site and find the ones ready for execution. That is, it tests if the unit is in the context or it is referenced by a an active process, and if for all the variables used by the unit there is a binding to a data unit set. In this case the unit is put in the `executable` list.
  - `handleRemote()` : to allow exchanging of messages across the sites. To be developed.
- `eval()` : it picks up a statement from the `executable` list and executes it calling the method `perform()` of the code unit selected.
  - `pick-up()` : select non deterministically a statement for execution for the `executable` list of code units.
  - `execute(CU c)` : calls the `perform()` method of the statement in the selected code unit.

There are some auxiliary methods like `bindDUDU()` , `bindDUCU()` , `bindDUGamma()` , `linkDuvars()` , and `linkCUvars()` .

9.3.3 The Handler

The Handler is called from the μ-Code server in order to unpack an incoming group. The method `unpack()` gets the element from the received group and put it in the `pending` queue.

9.4 Discussion and Future Work

Mobile agents systems have been mostly implemented in Java [WPM99], with some exceptions (i.e., Agent Tcl [Gra95], and Telescript [Whi96], Jocaml [CF99]). The choice of the programming language has often influenced the system design choices. The unit of mobility in the Java based mobile systems usually is an object, i.e., the executing agent. Code mobility in mobile systems is usually limited to fixed class loading mechanisms, most notably refinements of the Java class loader (like in Aglets [LO98], Java/RMI [RMI98], or Web Browsers). In LILLIPUT the unit of mobility is very fine-grained, however mobility of more complex structures, like processes, is allowed and class loading strategies are implemented with code unit migration. Moreover, data and code mobility are at the same level of the LILLIPUT abstraction. Therefore, nothing would prevent us from defining some sort of "object loaders", to migrate object status close to the code instead of the usual other way around.

LILLIPUT allows all the general mobile agents operations, such as classes and values sharing among processes (with the referencing mechanism), and replication of resources, code and agents. At a very fine-grained level (even single lines) LILLIPUT also permits code to be injected into the agent to increase its capabilities, or to replace obsolete behaviors. In Section 9.1 we showed an example of this, shipping and updating the "increment" strategy of a remote process.

There are many domains in which features of flexibility and dynamic reconfigurability are important, we are investigating LILLIPUT applications to domains such as active networks, where code needs to be transferred with packets to instruct routers about preferred routing strategies, or wireless networks, where small devices with scarce resources and connection need to interact and exchange code and data.

The model at the basis of LILLIPUT aims to be as general as possible. For instance referencing of units is allowed across hosts, and cross referencing between processes is permitted. Once the referencing process and the referenced unit are not on the same host the unit is not considered as part of the process scope (i.e., no binding is allowed to that unit). However, references are recorded so that when the process and the unit happen to be co-located again the binding is automatically permitted. In the LILLIPUT implementation we constrained the range for referencing. References are only only local

between processes and co-located units. Furthermore no cross referencing between processes units is allowed (i.e, a process can only reference units on the library of the host, thus simulating code/data library sharing). References are severed upon migration and no recording is kept. Nevertheless, we are investigating the advantages and drawbacks of the implementation (but also formalization) of more general referencing mechanisms, introducing referencing by type (instead of the used referencing by name), or allowing remote and permanent referencing à la Obliq [Car95].

LILLIPUT binding of variables and search of elements is based on names. Different strategies, possibly based on types, can be implemented as well. Given the fine-grained level and the range of primitives available in LILLIPUT many security issues rise. "Who has the right to do what on which element?" is the general question to summarize the many security problems that the approach has to face, and that have to be investigated. On-going work on the implementation of LILLIPUT also involve the development of the front-end of the system and the automatic mapping between Mobile UNITY and Java.

Research on possible languages that embody the decoupling between unit of execution and mobility have been carried on using XML [BPSM98a]. In [EMF00] an incremental migration of pieces of XML documents is used for application in the area of management of big numbers of workstations (i.e., behavioral update) and code distribution on thin clients: we describe this work in the next part of this thesis.

Chapter 10

Summary

Code mobility is generally perceived to take place at the level of agents and classes. The model presented in this part adopts an unusually fine level of granularity by considering the mobility of code fragments as small as single variables and statements. Our primary goal was to demonstrate the feasibility of specifying and reasoning about computations involving fine-grained mobility. Nevertheless the study has been instrumental in helping us develop a better understanding of basic mobility constructs and composition mechanisms needed to support such a paradigm. Composition and scoping emerged as key elements in the construction of complex units out of bits and pieces of code. The need for both containment and reference mechanisms was not in the least surprising given current experience with object-oriented programming languages but it was refreshing to rediscover it coming from a totally new perspective. The distinction between the units of definition, mobility, and execution proved to be very helpful in structuring our thinking about the design of highly dynamic systems. The necessity to provide some form of name service capability in the form of the find function appears to align very well with the current trend in distributed object processing. The resulting model shown in Chapter 7 is unique in its emphasis on verifiability and novel in its usage of cascading reactive statements, a construct akin to event processing but much more general. These features are, to a very large extent, the direct result of our attempt to reduce the programming notation we offer to the semantics of Mobile UNITY.

The granularity of the movement in process algebra is based on the notion of *process* (i.e., computation code) that is the actual unit of mobility. As we also said in Chapter 1, extension of this idea have been devised: Ambient calculus [CG00] and Seal calculus

[Vit99] introduce an explicit notion of *environment* (*ambients* in Ambient calculus and *seal* in Seal calculus). Environments define the computation scope. The unit of mobility is the environment that can carry computations and other environments while moving. Environments are unit of execution and of mobility at the same time as they rely on the notion of process common to the process algebra based languages. The approach presented in this part focuses on the decoupling on unit of mobility (i.e., the unit) and unit of execution (i.e., the process) in order to separate mobility related issues from more execution related issues like activation/deactivation status. We, therefore, provide specific operations (**activate**, **deactivate**, **terminate**) to act on the execution state of processes.

We can see possible future work in the security field also at this level of fine-grained mobility, especially in terms of resource access constrains: a first step could be adding operators constraining the visibility of units (as we discussed in Chapter 8).

Subjective and objective mobility of entities is an other security related issues, as it has to deal with access rights to entities: Ambient calculus is based on *subjective* mobility, i.e., every environment can decide to move with its content whenever it wants to, while Seal calculus prohibits this behavior for security reasons. In Seal calculus the environment decides on the movement of the contained entities (*objective* mobility). In our model the **move** construct is invoked in a code unit, that, to be executed should be part of an active process. The move can act on other entities, on the containing process, and on the code unit itself. We did not want to constrain the model assuming for instance that the move can only act on inactive processes, or it can only move local entities, or it cannot act on itself. All these constrains can be formalized on top generating a set of refined models that fit different purposes.

One of the main aims of the work presented in Chapter 7 and 8 is to provide basic operations for mobile code systems. All the formalisms considered provide more or less explicit mobility constructs: in π-calculus mobility is much more implicit than in Ambient calculus, though. Ambient calculus also provides an *open* operation able to dissolve the boundaries of an environment. Seal calculus does not provide that operation for security reasons. We do not provide an *open* operation as it can be built on top of the basic primitives of the model. In fact an open can be formalized as a movement (i.e., *move*) of all the contained entities of a process and of a *destroy* of the process itself. We specify basic movement operations for entities of different granularity (i.e., data, code, and processes).

The two cloning operations (i.e. *put* and *clone*) differ depending on initialization value of the copied entity. All the process algebra based models exploit the replication construct (.e., !) to formalize cloning. No notion of initialization values is provided.

Resources handling happen to be a fundamental issue in mobile code setting: we provide an operation to establish references to resources. Ambient and Seal calculus rely exclusively on the notion of scoping defined by the environment hierarchy for handling resources sharing: in our model processes act as containers and scope boundaries, however an explicit operation is provided (i.e., *reference*) to allow more general sharing that can be modeled by the designer.

In Chapter 9 we have presented a prototype of the fine-grained model, showing the implementability of the approach, in the next part we show how these ideas are embodied into an XML based approach. We will also describe some applications of the fine-grained approach.

Part III

Implementing Incremental Code Mobility

Chapter 11

Active Documents

In this part we use established tehcnologies to embody the fine-grained mobility of code strategies investigated in the previous chapters of this thesis.

In this chapter we introduce a recent technology (XML) and the way we have exploited it to be able to display formal notations on Internet Browsers. We show how the approach is used for formal notation display, in particular for Z [Spi92] documents display, enriching XML [BPSM98a] documents with customized code with "displaying power". the way in which the displaying is encoded and used is a first and simple attempt to add "activity" to XML documents. Active objects linked to the XML documents are loaded when needed in a very reconfigurable and flexible manner. Our first use of the approach was for displaying purposes. We describe this next.

11.1 Managing complex documents over the Internet

In the last years, we have seen the WWW being slowly transformed from an environment for sharing documents and data among members of specialized communities (be they scientific, research, artistic, social ones, etc.) to a general-purpose new medium for advertising and marketing commercial enterprises to the public at large. Since commercial use has a larger impact and therefore power on the advances of the medium, the specific needs of specialized communities have been overlooked in the further development and advances of this new medium.

For instance, the use of the WWW as an environment for software design introduces new problems and challenges: the use of the WWW to support software process workflows,

sharing specification documents, allowing to read and write them, and providing hypertextual links among documents is felt as a hot topic [KDJY97, SN97], but little specific aid to software designers is available on the WWW at large.

A very important need that many communities of engineers have is the support for special notations that are current or even absolutely necessary within that community. Currently the Web is very poor in supporting special notations. The typographical rendering of WWW documents is usually defined using the HTML mark up language; currently, it is the basis of most intranet document management systems [Ban97, Res97]. In its many versions, HTML provides textual support for elements such as input fields, buttons, choice lists, etc. along with structural and formatting commands for text within the data format of network documents, and, of course, the hyperlinking capabilities that gave it its name.

It has been extremely important that HTML allowed both complex interfaces and proper and traditional text content to be described in ASCII-based source documents. HTML has shown the way that text-based support for non-textual content eases understanding, tool creation and debugging of applications that deal with it. Furthermore, they allow a complete intermix of different concerns, such as interface elements and text characteristics, thereby fostering the creation of complex interfaces that are at the same time rich in content and sophisticated in their interaction with the user.

On the other hand, HTML is limited in that it has only a small set of allowable elements, that is, only those that are explicitly defined in the standard. Whenever some authors' needs exceed the capabilities of the elements already defined in HTML, a different approach needs to be used: either the existing tags are abused for a different purpose than that for which they were designed, or an image is used, or a Java applet is created providing the desired functionality.

These kludges have obvious and well known drawbacks, that have lead to the development of many alternative (and partial) solutions. For instance, Cascading Style Sheets (CSS [LB97, BLLJ98]) allow authors to separate the efforts to specify special graphic effects and the structure and determination of the actual content of the document, allowing complex typographical rendering to be built on top of still readable plain HTML documents. XML [BPSM97] is another tentative in that direction: instead of forcing authors to the limited and closed set of pre-defined elements, XML is a meta-markup language that

allow authors to define their own sets of markup elements that are most appropriate to the specific class of documents they are dealing with. Adjunct languages (XSL, XPointer, XLink [MD98]) are used by authors to associate these elements to some rendering or linking semantics for their display on paper or screen. This allows a definitive separation between the description of structures and roles of the documents and the description of their graphical rendering on a computer terminal or on a high resolution printer.

Neither solution is currently completely satisfying for supporting specialized notations because both are only concerned with supporting text-oriented content only. Many notations have sophisticated need that go well beyond texts. For instance, specification languages like Z [Spi92] are often based on specialized notations (mathematics and logic symbols): it would be useful to be able to give a visual interpretation of these symbols and to allow them to be displayed on WWW pages.

The purpose of this section is to report on a Java rendering engine for XML data that we have implemented. The engine allows standard typographical support for text-oriented XML documents, as well as extensible graphical support for additional needs, in particular for specialized notations. We have created a complete graphical and typographical support for formal specification documents written in Z. The rendering engine we are describing works as a completely autonomous applet inside unmodified Java-enabled browsers such as Netscape Communicator or Microsoft Internet Explorer.

11.2 Creating Z specifications

Several tools exist to this date to help software designers to write, test, and share documents containing their Z specifications. A complete guide to all the existing tools for Z can be found in the site `http://www.comlab.ox.ac.uk/archive/z.html`.

We can divide the available tools into four main categories: fonts, browsers, editors, and type checkers.

True Type fonts for Z are available to use with common word processors on many platforms including Windows and Macintosh, but fonts of course only give access to the special mathematical characters of the Z language, forcing users to use non-specific features of available tools to create the graphic boxes of schemata and other Z elements.

Customizable formatters such as LaTeX [Lam86] are the most common tools to write Z specifications. General style files for LaTeX, such as `oz.sty, fuzz.sty, ztc.sty`, have been published to precisely render Z specifications.

Logica has created a syntax-driven WYSIWYG editor for Z on MS Windows platforms. Such an editor also integrates a type checker and forces the production of well-formed Z specifications by providing facilities for building, editing, checking, and viewing Z specification documents. Being WYSIWYG, the editor can display the Z constructs and symbols as they would appear on a printed page.

The paper [M+95] describes the Z Browser, an application for displaying Z specifications running on MS Windows. Such a tool is aimed at Z novices, and is integrated with a complete help system for Z grammar and notation, thus it supports the construction, syntactical check, and visual layout of Z documents.

Several analysis tools also exist for Z specifications. For instance, CadiZ [Jor91] is an integrated suite of tools for creating Z documents. It understands source files in LaTeX and Word for Windows, and can visualize implicit Z expressions (i.e. schema calculi) by showing their expansions.

Finally, the ZTC [Jia94] type checker accepts LaTeX-formatted Z specifications as well as text-based ones. ZTC also suggests using a special syntax based on concatenation of ASCII characters for mathematical symbols.

In summary, it is clear that Z is a highly structured notation both graphically and semantically complex, and that writing, checking, and displaying Z specification documents is yet an unsolved issue.

11.2.1 Hypertext and Z specifications

There are several good reasons to provide hypertext functionalities to Z specifications. A complex specification is intrinsically composed of many connected chunks (schemas, etc.) that refer to each other in a peculiar, often unpredictable way. Furthermore, the idea of literate programming [Knu84] requires that schemas and texts interleave freely, so that the reader is provided with a narrative explanation of the most complex schemas, and a formalized and exact specification of vaguer descriptions. These remarks naturally call for a hypertext solution.

Moreover, collaboration and sharing are even better reasons for providing hypertext support to Z specifications: formal specifications are but one step in the complex process of system design, verification, and implementation [FKV94]. Modern development processes are enacted by teams of people that cooperate, interact, and discuss. Being able to create, access, and verify formal specifications within the usual tools of our everyday work, publish them, connect them to the other deliverables of the design and implementation processes would allow a tighter integration between formal design and actual implementation [CFR97].

Till recently, Z specifications could only be visualized on the WWW by creating images in one of the supported inline formats, such as GIF. This leads to a very cumbersome and unnatural creation process, since the Z specifications have to be created in a different environment than the text, and furthermore non-specialized graphic editors have to be used and restrained in order to produce graphically acceptable schemas. It is also a very unnatural and clumsy way of accessing to the information: an image of a schema is a completely opaque object, where the subparts, the texts, the formulas are completely inaccessible; it is a bitmap that cannot be further processed because the content and meaning have been lost: the content of a schema cannot be searched, the specifications cannot be indexed, analyzed or verified.

A first attempt to show Z specifications on the WWW was described in [MAS97], designing a plug-in for Netscape and Internet Explorer that accepts Z specifications written using one of the existing LaTeX styles.

Although this approach is very original it has two main limitations: first, visualizing Z documents requires the availability of the plug-in, which is architecture-dependent (it only exists for MS Windows). Secondly, the LaTeX format is alien to the available SGML-based formats suggested for the WWW: in fact, writing Z schemas in LaTeX requires a different syntax and approach than writing the surrounding free-flow text in HTML, and the specifications live independently of the host document. The first problem has been addressed: the Z browser is becoming a Java applet, which is architecture-independent and can be run on most computers of the current generation.

J. Bowen and others in Reading are working on a Java applet to visualize Z schemas [BC98]. Our approach, detailed in section 11.5, is related but with relevant differences.

11.2.2 The advantages of markup languages

HTML has been extremely successful in allowing unsophisticated network users to become authors of fairly complex documents, even in the absence of widespread editing tools. Nonetheless, there has been in the past two or three years a widespread awareness ([SMG94]) that HTML has reached its potential, and that a change of paradigm was necessary.

The major drawback of HTML is that it allows only a pre-specified set of elements. Authors can only use these elements, and have to limit their authoring needs to what is available within the existing language, or to force these elements beyond their intended meaning.

HTML is an application of the Standard Generalized Markup Language [SMG94], that is, a class of documents conforming to the SGML Document Type Definition (DTD) that describes "HTML documents". SGML, being a meta-language describing classes of documents rather than one specific class, is free of the above mentioned limitations of HTML: by appropriately creating a custom class of documents, and defining the legal elements therein, authors can provide support for any kind of rhetoric need, however complex and arcane.

Unfortunately, SGML is considerably more complex to learn and design documents with than HTML, and it has been felt that this would prevent its generalized adoption. Therefore the SGML working group of the Word Wide Web Consortium was asked to develop a new mark-up meta-language, namely the Extensible Markup Language (XML) [BPSM97], to take the place of SGML on the Web. XML documents would have to be straightforwardly usable over the Internet, compatible with SGML, and easy to create.

There are several standards being developed within the XML framework: the most important is XML itself, a meta-markup language that allows user to create their own set of elements for their class of documents. XPointer and XLink [MD98] extend HTML linking mechanism by providing external specifications of locations, multiple links, external links, etc. XSL [CD98] associates rendering behavior (e.g. character and paragraph settings) to XML elements through a mapping and rewriting language. XML-Namespace [BHL98] allows elements coming from different namespaces (document types, for instance) to live together in the same document. Very important is MathML [IM98], a markup language for mathematics, formerly part of the unborn HTML 3.0 and subsequently detached in

an autonomous standard finally converted into XML. MathML covers most needs for mathematical rendering, and is capable of showing most of the strange glyphs that are part of the Z language, but is not thought for Z and does not provide support for other, more specific needs of the Z notation.

Interestingly, in the Z community an SGML-based language for Z specifications already exists: the Z Interchange Format (ZIF for short) [BN92] defines a portable representation of Z, that can be used by all tools supporting SGML. The ZIF is basically a Document Type Definition (DTD), namely an SGML specification defining the syntax of documents that contain Z specifications. In [GC95] a study of the usage of the ZIF was presented, according to which ZIF can be fruitfully used to create editors for Z documents using standard SGML tools, and that Z specifications encoded using ZIF could easily be included in other SGML documents.

XML documents are valid SGML documents. Most existing SGML DTDs can be used with no modifications in an XML environment. Notably, the Z Interchange Format is one of such DTDs.

It is therefore possible to use the definitions specified in the ZIF within XML tools, in order to create web-friendly visualizations of Z specifications. Alternatively, XML tools allow the HTML tag set to be described and extended as needed. By joining the HTML DTD with the ZIF DTD, and producing a capable browser, it is possible to write HTML documents that contain Z specifications as markup items, instead of images, thereby keeping all the useful properties that markup has over bitmaps.

In this chapter we report about one such tool, that allow the display of text-based XML documents enriched with Z specifications. This mechanism can obviously be extended to handle the display of any kind of notation within a XML document.

11.3 Displets and markup languages

Displets were proposed in [VCB97] as a way to extend HTML documents using Java. The HTML language was extended on a per-document basis by defining new tags as needed, and providing Java classes to take care of their graphical display. While not providing all the functionality and flexibility of a full meta-mark up language such as XML (Sect. 11.2), HTML extended with displets could allow all kinds of specialized notations and graphical

effects while at the same time leveraging over the existing and well-known set of elements defined by HTML.

Our first experiment with rendering arbitrary, non-text-based mark up extensions [VCB97] was to modify an existing browser to allow the parsing and the visualization of new HTML-like elements. To do so, we took an early version of the HotJava browser, whose source code was freely available, and modified it so that it could accept on-the-fly extensions of the HTML DTD and load the appropriate classes (called *displets*) whenever the newly defined tags were to be displayed. That experiment was extremely limited, in that we used an old version of the Java language, and worked only on a specific version of a specific browser. Furthermore, we heavily relied on the existing rendering architecture of the browser and just provided a minimal effort implementation (basically a displet was just a sequence of drawing instructions for the visualization of the elements).

In [CRV98], on the other hand, we reported about the DispletManager applet, a general, extensible rendering and architecture we have been working on, which can be used for both extensions to HTML and straight XML documents. This architecture is embodied in a Java applet that can be run within any Java-enabled browser such as Netscape Communicator or MS Internet Explorer.

Fundamental design requirements for the rendering engine have been:

- it must be possible to create special code for rendering arbitrarily odd data types, in particular non-textual data (*displets*).
- all displets must easily integrate with each other: a chart element may have a mathematical formula as one of the labels, and some staff notation as another, where some notes may act as hypertext links.
- the rendering engine must work both for extended HTML and for straight XML, and the displet classes must be identical.

Figure 11.1 shows the general structure of the DispletManager applet:

The document chunk to be displayed, be it HTML or XML, is loaded by the displet manager and parsed by the appropriate parser. The resulting tree is then recursively (depth-first) analyzed: the appropriate displet classes are activated to create the rendering (i.e., the display object) of their element on the basis of the rendering of their sub-elements.

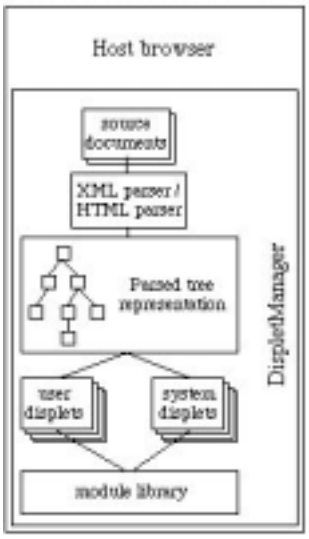

Figure 11.1: The general structure of the DispletManager applet

No class is allowed direct access to the screen: on the contrary, each displet creates a (set of) off-screen bitmap(s) that its ancestor can pass, ignore, modify or add to.

Several specialized browsers exist for XML-based special notations. For instance, WebEQ for MathML [Web] and Jumbo for CML, a XML-based notation for chemical data [Jum]. Although a specialized browser would have probably been more efficient and sophisticated for Z elements, too, we felt that a general rendering engines for all kinds of notation was preferred, leading us to a more general and extensible architecture for Z and other needs.

11.3.1 Applying displets to XML documents

The XML language allows authors to define their own set of elements (tags) to structure and organize their documents. Of course these elements do not have a pre-defined meaning, nor even a pre-defined visualization. For instance, while it is known that the "H1" element in HTML has both a structural role (the heading of a first level section) and a graphical rendering (use a large font and align it on the left), a corresponding "major-heading" element in a XML document would have no machine-understandable structural role (but this is not a problem), nor a known graphical rendering (we can not even determine whether the element is a block, a paragraph or an inline element).

The XSL [CD98] language is used for associating rendering information to an XML document. Each XML document that needs to be displayed on screen or on a printer would have a XSL document associated. The XSL document contains a series of "rules" mapping the XML elements of the document to one or more *flow objects* (i.e., graphical objects such as blocks, paragraphs and inline texts).

Although in XSL the set of available flow objects is fixed, we allow the specification of new flow objects, that can be specified in the rules just like the standard ones. Each flow object corresponds directly to a displet class.

What follows is an example of a simple XML document contained in the DispletManager applet and its associated XSL style rules. The style sheet refers to two flow objects: a standard paragraph object (belonging to the CSS family of flow objects available within the standard XSL proposal), and a special "reverse" flow object that is prepared as a displet by the author of the document:

```
<applet code="DispletManager.class" width=500 height=200>
<param name = "style" value = "

<xsl>
<rule>
        <target-element value='para'/>
        <css.div font-size='12'>
                <children/>
        </css.div>
</rule>

<rule>
        <target-element value='rev'/>
        <example.reverse>
                <children/>
        </example.reverse>
</rule>

</xsl>  ">

<param name = "XMLcode" value="
```

```
<para>This is an example of a text rendered in
<rev>reverse</rev></para>

">
</applet>
```

The DispletManager applet for XML has two arguments: the first contains the style sheet document according to the XSL rules, while the second one contains the XML document that has to be displayed, using the elements that are described in the XSL style sheet associated.

Upon loading the applet, the displet manager will start the XSL engine and read in the 'style' parameter. This is parsed (by a XML parser, because it is itself a XML document) and organized. Then the "XMLcode" parameter is read and parsed by the same XML parser, creating a tree of elements and data.

The XSL engine will then match each element in the XML document with the *pattern* contained in each XSL rule. When the most suitable match has been found, the rest of the rule (the *action* part) is executed, creating the flow objects listed and feeding them their content (usually the rendering of their subelements, as specified by the `<children>` tag).

In this example, the 'para' element of the XML document matches the first XSL rule, triggering the creation of a 'div' object of the standard CSS package (a paragraph) with a specific parameter (font-size=12), fed with the children of the element (i.e., the words and the elements contained within the para tags). Then the 'rev' element is considered, and matched to the second rule of the stylesheet, triggering the creation of a 'reverse' object belonging to the 'example' package, fed with its content. As soon as the rendering of its content has been readied (by creating the necessary bitmaps), the displet class corresponding to the flow object is activated.

Each displet will then produce a (list of) bitmaps of its content. For instance, the 'div' displet of the CSS package will set a few parameters (such as margins, line spacing, font, and size) that may affect its sub-elements, wait for the XSL engine to return control after its content has been readied, and build its own content by combining the bitmaps of each

Figure 11.2: Rendering a simple displet

word into lines according to the given constraints. Figure 11.2 shows the above mentioned document results on screen:

11.4 The Rendering Engine

The rendering engine used by the DispletManager applet consists of a set of Java classes that provide the rendering for the appropriate document elements. These classes are all subclasses of the DocElement class, which provides the framework of the rendering procedure.

All classes provide a `createBitmap()` method, whose purpose is to create and return the bitmap of the flow object of the considered mark up element on the basis of the bitmaps of its sub-elements. The `createBitmap()` method is usually not seen by the implementer of new classes, and provides the following functionalities:

- an active drawing environment is managed. The drawing environment is a set of parameters that are used by the rendering methods of the classes in order to decide how to create the bitmaps. For instance, a paragraph-like class may set some parameters that will be used by itself, such as margins, line spacing, alignment, etc., and some that will be used by its sub-elements, such as font name, font size, font color, etc. The `createBitmap()` method allows a displet to set its own attributes with the `setParams()` method, and restores the previous situation when the displet

is finished. Since `createBitmap()` methods are recursively activated, this creates a stack that provides the proper parameters at any level of recursion.

- the rendering of sub-elements is managed. The presence/absence of the element in the XSL rule may cause or prevent the rendering of the sub-elements of the current element.

- the rendering of the element is managed. After the bitmaps of the sub-elements of the element have been created (if appropriate), the `createBitmap()` method calls the `render()` method, which in turn creates the final bitmap (or set of bitmaps) that will be returned. Different classes will implement `render()` differently: for instance, the `render()` method of a block element will collect the bitmaps of its sub-elements in a vertical stack (one above the other), and provide a single bitmap of the whole element, while the `render()` method of a paragraph will collect its sub-elements side-by-side in lines of the given width, and provide a bitmap for every line it has created; this allow the element containing the paragraph to decide how much of the paragraph to display at a time (for instance, in case of scrolling).

- active elements are specified and created. Active elements are those that will need to react to user and system events after they have been displayed. For instance, form elements and anchors have an associated behavior that is activated when the user selects them.

Figure 11.3 shows the inheritance structure of the classes of the module library:

DocElements can either be data, entities or tag elements. DataElement classes are used for the content of mark up elements, i.e., #PCDATA in SGML and XML DTDs. They can either be text or hidden elements. EntityElements are provided for the management of XML and HTML entities such as & or the definition of new ones. TagElements are used for the creation of the structure flow objects of the document: they are either flow objects, block objects, inline elements or special elements.

- A block element is a single object that stands alone in the vertical layout of the document. Paragraphs or tables are block elements. A flow element is a block element that is built piecemeal: while plain block elements are built from start to end before the `createBitmap()` returns, flow elements build each of their sub-element

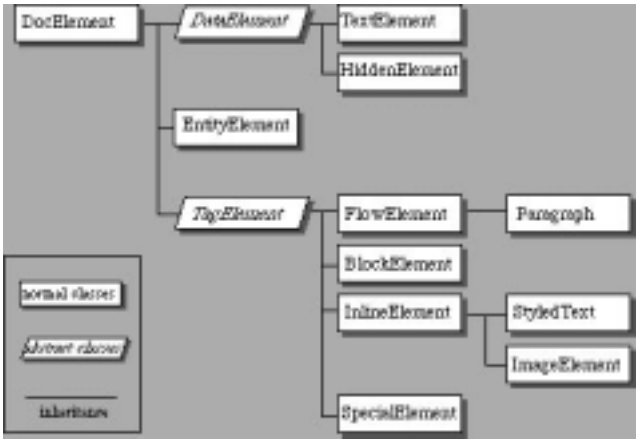

Figure 11.3: The inheritance structure of the module library

and return, and are called as many times as there are sub-elements. This allows long and complex elements to be rendered only for the possibly small section that is actually displayed. For instance, HTML and BODY are considered flow elements, so that the display of an HTML document can start as soon as the first object is completed, and be interrupted when the available display space is filled.

- Inline elements are elements that can be put side by side with their siblings. Inline elements are used within block elements and may be text-based, images or something else. The StyledText class allows the specification of text runs of arbitrary styles. Inline elements specify the places where they can be broken by creating as many bitmaps as break points. This allows the containing paragraph or block element to determine where the line should be broken.

- Special elements are completely tailorable. While in the previous classes displet programmers can only overload the `setParams()` and `render()` methods, here all methods are overloadable, and can be customized.

As an example, this is the complete source code of the 'reverse' displet:

```
package example;
import displet.*;

public class reverse extends StyledText {
```

```
        public void setParams(StyledTextParams p) {
                Color c = p.fgColor;
                p.fgColor = p.bgColor ;
                p.bgColor = c ;
        }
}
```

The reverse displet is a subclass of the StyledText, which is a subclass of the InlineElement class. These are classes for text-based objects that behave as in-line elements (eg. bold, italic, etc.). As it can be seen, the programmer of such a displet only has had to specify a parameter and have the `render()` method of its superclass handle all the details. The displets for showing Z specifications are shown in the following section.

11.5 The Z browser

The main extension to HTML we have considered using displets is the implementation of the complete ZIF DTD. Authors writing Z specifications can create documents containing their Z specifications in a markup language similar to HTML and completely intermixable with plain text and other HTML features such as links, tables, etc.

The ZIF format defines several elements (tags) for the building blocks of the language, such as schemas, definitions, etc., and several entities (literal macros) for the special characters inherited from mathematics and logics. Each element is implemented by a displet that creates a bitmap where the content of the element is appropriately composed and the graphical elements such as boxes, lines, etc. are then added. Entities on the other hand are elements of a graphical alphabet that is contained in a single GIF image and is loaded with the displets.

The following is an example of a Z schema using the Z Interchange format:

```
<givendef>
    NAME, DATE
</givendef>
<schemadef>
    BirthdayBook
    <decpart>
        <declaration> known: &pset; NAME</declaration>
```

```
        <declaration> birthday: NAME &pfun; DATE </declaration>
    </decpart>
    <axpart>
        <predicate>known = &dom; birthday</predicate>
    </axpart>
</schemadef>
```

A schema is defined by a tag called schemadef, which contains three elements: the name of the schema, a declaration part and an axiom part. The declaration part contains one or more declarations, and the axiom part contains zero or more predicates. Appropriate ordering and nesting of elements is enforced by the DTD, and is checked when parsing the document. The notations "&pset;", "&pfun;" and "&dom;" are three entities (respectively, the partial set symbol, the partial function symbol and the domain symbol) that will be substituted by the corresponding element in the graphical alphabet containing all the relevant Z symbols. The displet manager can appropriately show document bits as the previous one in a WWW browser.

Since many Z specifiers use LaTeX to produce their Z documents, we have developed an off-line translator called "Zed2XML" that transforms Z specifications written in LaTeX using style `oz.tex` into a corresponding HTML document with the appropriate extension.

For instance, given the following Z specification (the basic birthday book example from [Spi92]):

corresponding to the following LaTeX source document:

```
\documentclass[italian,12pt,twoside,openright]{report}
\usepackage{amsfonts}
\usepackage{oz}

\begin{document}

\begin{zed}
[NAME, DATE]
\end{zed}

\begin{schema}{BirthdayBook}
known: \power NAME\\
```

```
birthday: NAME \pfun DATE
\where
known = \dom birthday
\end{schema}
\end{document}
```

The Zed2XML application transforms the previous LaTeX example in the corresponding applet specification:

```
<applet  archive="displet.zip" code="XMLManager.class"  width=450 height=200>
<param name = "XMLcode" value="

<givendef>
    <a name="name">NAME</a>,
    <a name="date">DATE</a>
</givendef>
<schemadef>
    BirthdayBook
    <decpart>
        <declaration>
            known: pset; <a href="#name">NAME</a>
        </declaration>
        <declaration>
            birthday: <a href="#name">NAME</a> pfun;<a href="#date"> DATE</a>
        </declaration>
    </decpart>
    <axpart>
        <predicate>
            known = dom; birthday
        </predicate>
    </axpart>
</schemadef>

"> <param name = "style" value="
<xsl>
    <import name="htmlcss.stl"/>
```

```
    <import name="Zpackage.stl"/>
</xsl>

"></applet>
```

The output of Zed2XML is the HTML specification of the *DispletManager* applet. As it can be seen, we are following the ZIF format quite strictly. For the sake of brevity and reusability, standard stylesheets are used and invoked by a simple import command in the specification of the applet.

The 'htmlcss.stl' document contains the XSL rules to use HTML elements within XML documents. For instance, we are using here HTML links with the A tag. This is the relevant excerpt from the 'htmlcss.stl' document:

```
<rule>
    <target-element type="a">
        <attribute name="href" value="%2"/>
    </target-element>
    <css.a href="%2">
        <children/>
    </css.a>
</rule>
```

The 'Zpackage.xsl' document contains the XSL rules to use the Z displets within XML documents. This is an excerpt from this stylesheet:

```
<rule>
    <target-element type="givendef"/>
    <zpack.givendef>
        <children/>
    </zpack.givendef>
</rule>

<rule>
    <target-element type="schemadef">
    <zpack.schemadef>
        <children/>
    </zpack.schemadef>
```

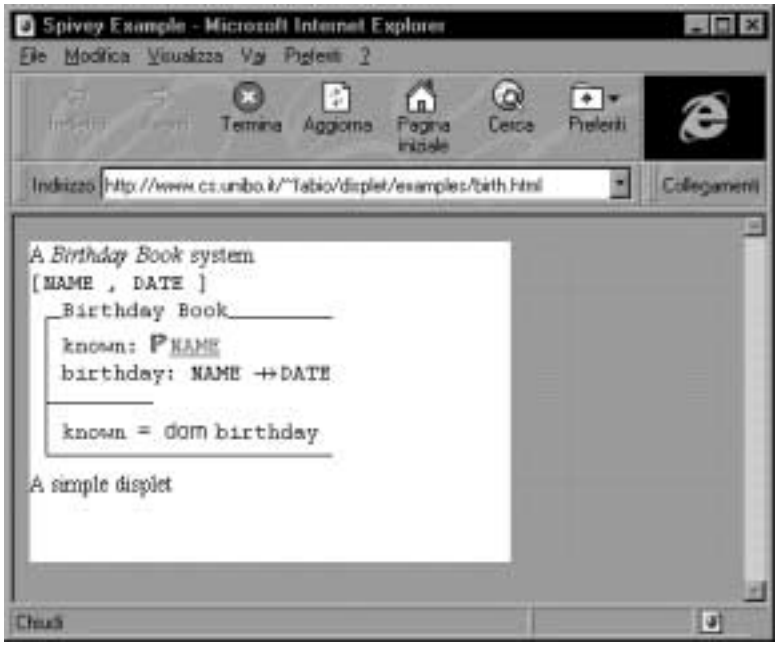

Figure 11.4: Visualization on the WWW of a Z schema

```
</rule>
```

When run on a WWW browser, the previous documents is shown as in Figure 11.4.

We remark that all displets integrate with each other and can refer to each other freely. In our case, the Z schema contains a hypertext link described in a different package. Z elements and plain text based XML elements freely intermix: it is possible to put standard HTML tags within Z schemas, for instance an author may require that some declarations of a schema are written in bold. The Zed2XML translator automatically connects types used in declarations to their definitions using plain HTML links. The author may freely add or modify the available links and HTML features, and include additional HTML elements, as well as native XML elements or elements belonging from other packages of displets.

In the next chapter we show how the XML based approach allow very flexible handling of code and data bindings.

Chapter 12

Incremental Code Mobility and Applications

The increasing popularity of Java and the spread of Web-based technologies are contributing to a growing interest in dynamic and reconfigurable distributed system architectures. Such reconfiguration can be achieved with code mobility, transferring fragments of code across the network, from one host to another. Code mobility is generally associated with Java, where byte code representations of classes can be loaded from remote hosts.

The potential mobility range is however wider, starting from simple data mobility, where information is transferred. Simple examples are the actual parameters that are passed to a remote procedure call or the web page that is returned to a get request in the HTTP protocol. At a level above this, code mobility allows the migration of executable code: browsers loading applet classes from remote servers are very common examples of code mobility. Java-based technologies, for instance, Java RMI [RMI98] and Java Virtual Machines, such as those built into Web browsers, offer a mobility granularity at a class level. Mobile agents [WPM99], in which code and data move together, can be considered the highest level of mobility that can be achieved in a logical context.

Several application domains need a more flexible approach to code mobility then can be achieved with Java. This flexibility can either be required as a result of low network bandwidth or scalability. The 9,600 baud bandwidth between a server and a GSM mobile phone cannot cope with downloading large amounts of Java byte code from a server. Scalability requirements can mean for example, that applications on several thousand clients have to be kept in sync or that tasks are so computationally intensive that they need to be distributed across multiple processors. These processors need to be instructed in a flexible way.

In this chapter we show how to achieve more fine-grained mobility than in the approaches that are based on Java. We demonstrate that the unit of mobility can be decomposed from an agent or class level, if necessary, down to the level of individual statements. We can then support incremental insertion or substitution of, possibly small, code fragments and open new application areas for code mobility such as management of applications on mobile thin clients, for example wireless connected personal digital assistants (PDAs), user interface construction and inconsistency management.

This work builds on the formal foundation for fine-grained code mobility that we presented earlier in Part II. There we develop a theoretical model for fine-grained mobility at the level of single statements or variables and argues that the potential of code mobility is somehow hidden behind the capability of the most commonly used language for code mobility, i.e., Java. In this chapter, we share that vision and focus on an implementation of fine-grained mobility using standardized and widely available technology.

In this chapter we give a description of how to use the eXtensible Markup Language (XML) [BPSM98a] to achieve flexible, fine-grained and incremental code mobility. In Section 12.1, we discuss related work, most notably XML and other approaches to logical code mobility. In Section 12.2, we show how XML supports the definition of high-level languages and how incremental code mobility can be defined with XML. In Section 12.3 we demonstrate how the implementation of mobile code systems supported by off-the-shelf XML products. The construction of interpreters for high-level languages is simplified by XML parsers and the Document Object Model (DOM) [ABC+98]. XML parsers construct abstract syntax trees of the XML document and the DOM standardizes an interface for traversals through abstract syntax trees. Section 12.4 we argue that fine-grained mobility has the potential for a set of application areas such as consistency management in distributed documents, user interface development, and management of applications on mobile thin clients. We give examples of the application of our approach in these areas. Section 12.5 evaluates the approach and identifies strengths and weaknesses.

12.1 Overview of XML and Logical Mobility

Physical mobility is concerned with the physical movement of hosts, such as notebooks, PDAs, mobile phones and wearable computers. Logical mobility is the ability to transfer

data and/or code from one host to another by using a network. This chapter focuses on logical mobility, though the approach is also applicable to information that transits between physically mobile hosts; in fact, Section 12.4 discusses how our work can be applied to manage applications deployed on PDAs. Logical mobility encompasses data and code mobility.

Data mobility is a very common mechanism and often used to exchange or spread information among different hosts distributed on a network. Data mobility can be achieved by passing parameters to remote procedure calls, object requests or the put and get operations of the file transfer protocol. With the introduction of the Internet and the World-Wide-Web the Hyper Text Markup Language (HTML) has been used as the predominant format for data that moves between hosts on the Internet.

XML [BPSM98a] is the next generation markup language for the Internet. XML is a subset of the Standard Generalized Mark-up Language (SGML) [ISO86]. Unlike HTML both XML and SGML allow users to define their own set of mark-up tags for structuring documents. These user-defined mark-up tags are defined in document type definitions (DTDs). A DTD is a context free grammar that defines the syntax of documents. XML documents always declare a reference to their DTD in order to enable generic parsers to obtain the specification of the grammar. Thus with the advent of XML, different formats for transferable data can be defined. Many different DTDs have been standardized to encode specific notations in XML. An example of a software engineering application is the XMI [OMG98b] that defines a DTD that can represent any UML [BJR99] model.

XML is not only useful for publication of documents on the World-Wide-Web, but that it can also be used as an application-specific transport protocol in distributed system construction. In [ESF99] the authors report about the use of XML for the transport of data between different distributed and heterogeneous components of a financial trading system. That system uses XML documents as parameters to CORBA object requests. Moreover, the OMG have requested proposals for the interoperability between their Interface Definition Language and XML [OMG99] that will address the seamless interchange of XML documents and equivalent complex values of IDL data types.

Data and code mobility in Java are supported through object serialization and class loading. The status of objects can be serialized and transferred from one host to another while the class loading strategies can vary, depending on the application. For instance,

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <!ELEMENT KarelProgram (turnon|go|turnleft|
         pickbeeper|putbeeper|turnoff|times)*>
  <!ELEMENT turnon EMPTY>
  <!ELEMENT go EMPTY>
  <!ELEMENT turnleft EMPTY>
  <!ELEMENT pickbeeper EMPTY>
  <!ELEMENT putbeeper EMPTY>
  <!ELEMENT turnoff EMPTY>
  <!ELEMENT times (turnon|go|turnleft|
         pickbeeper|putbeeper|turnoff|times)*>
  <!ATTLIST times howoften CDATA #REQUIRED>
```

Figure 12.1: The DTD for Karel's Instruction Set.

the Netscape class loader downloads applet classes from the web server of the containing HTML page; the Java RMI class loader allows the application to download the classes of the objects remotely passed as parameters at run time. The class of the moved object can migrate onto the new host or it can be fetched from a remote server. Many different technologies have been built on top of these simple mechanisms.

Mobile agents are such a technology. Mobile agents are autonomous objects carrying their state and code that proactively move across the network. Many new systems have been developed to support mobile agents [KZ97]. Agent mobility requires the migration of both code and state of the agent at the same time and they can move proactively performing tasks on behalf of users. The more recent mobile agents technologies are usually Java based [WPM99] however some examples exist of non-Java based mobile agents (e.g., Emerald [LHM88]).

12.2 Specifying Incremental Code Mobility with XML

XML provides a flexible approach to describe data structures. We now show that XML can also be used to describe code. XML DTDs are, in fact, very similar to attribute grammars [Knu68]. Each element of an XML DTD corresponds to a production of a grammar. The contents of the element define the right-hand side of the production. Contents can be

```
<?xml version="1.0"?>
<!DOCTYPE KarelProgram SYSTEM "karel.dtd">
<KarelProgram>
  <turnon/>
  <times howoften="2">
    <turnleft/>
    <times howoften="2">
    <go/>
    </times>
  </times>
  <turnoff/>
</KarelProgram>
```

Figure 12.2: An XML program for Karel.

declared as enumerations of further elements, element sequences or element alternatives. These give the same expressive power to DTDs as BNFs have for context free grammars. The markup tags of DTDs define terminal symbols. Elements of XML DTDs can be attributed. These attributes can be used to store the value of identifiers, constants or static semantic information, such as symbol tables and static types. Thus, XML DTDs can be used to define the abstract syntax of programming languages. We refer to documents that are instances of such DTDs as XML programs. XML programs can be interpreted and in Section 12.3 we discuss how such interpreters can be constructed using XML parsers. When such instances are sent from one host to another we effectively achieve code mobility.

In order to demonstrate these ideas, we consider a very simple programming language to instruct Karel, the robot. The language has first been defined in [PRS94]. In this chapter we only consider a subset of it for reasons of brevity. Karel's language has a set of primitives. These include `turnon` , to switch the robot on, `go` to make it proceed one step into its current direction, `turnleft` to change the robots current direction by turning left, `pickbeeper` and `putbeeper` to get and dispose of beeper objects and `turnoff` to turn Karel off. Moreover, Karel's programming language includes a number of control structures for repetition and conditional execution. Here, we only consider the `times` statement. It repeats a cycle of commands for a given number of times. Figure 12.1 shows the syntax of the subset of Karel's programming language defined as an XML DTD.

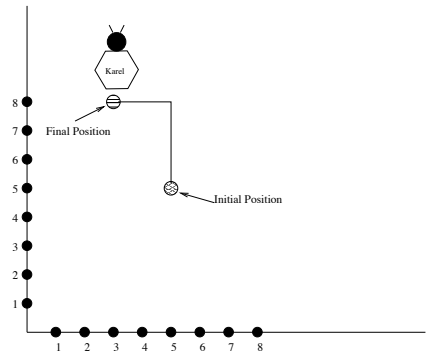

Figure 12.3: The actions of the robot.

```
<?xml version="1.0"?>
<!DOCTYPE times SYSTEM "karel.dtd">
  <times howoften="3">
    <turnleft/>
  </times>
```

Figure 12.4: XML Code Increment.

Figure 12.2 shows an instance of the DTD in Figure 12.1. This instance is a Karel program that instructs Karel first to turn left, then to proceed two steps, turn left again and proceed two more steps. Karel's route is shown in Figure 12.3. If we imagine that Karel is a real robot, that is instructed from some control host by sending these XML programs via, for example, a radio network, we have then achieved logical code mobility with XML.

Unlike Java code, which is sent in a compiled form, XML code is transferred as source code and then interpreted on a remote host. Unlike Java, XML does not confine us to sent coarse-grained units of code; XML documents do not need to begin with the root of the DTD, they can also start with other symbols of the grammar. This enables us to specify sub-programs and even individual statements. We refer to such code fragments as XML program increments. Hence, we can specify complete programs as well as arbitrarily fine-grained increments in XML.

Figure 12.4 shows such a fine-grained program increment. We can imagine that we want to change the behaviour of Karel by replacing the `turnleft` statement with this

increment and thus change the behaviour of Karel making it turn right instead of left [1]. As Karel is controlled by a slow radio network, we want to avoid re-sending the whole program but rather incrementally send the new program fragments.

The question that arises is how we specify the insertion or replacement of program increments. Addressing particular locations in an HTML document is achieved by "anchors". These anchors, however, cannot be defined by users who do not have control over the document. Likewise in our approach, the sender of an increment does not have control over the program once it has been sent and we cannot assume that programmers identify anchors or other labels a-posteriori that could then later be used for incremental code insertion or replacements.

To solve this problem, we use XPointer, an XML-related standard. XPointer is part of the XLink specification [MD98] and overcomes the limitation of HTML by supporting navigations within XML documents. These navigations are capable of addressing every document component without having to modify the document itself. We use XPointer to identify that component of an existing XML program that we want to replace with a new increment.

Going back to our example, Figure 12.6 shows an XPointer expression that determines the Karel program statement that we want to replace. The XPointer expression starts from the root of the program and then selects the first statement of type `times` , and in that statement it selects the `turnleft` statement. Thus, by specifying a fragment of a program in XML together with an XPointer expression, we can express incremental code mobility. Figure 12.5 shows how Karel's behaviour will differ after the new increment has replaced the `turnleft` statement.

We have so far shown how we can use XML to define programs and how we can define the update of code in an incremental fashion. In the next section we describe how we can utilize off-the-shelf XML technology in order to implement interpreters for application specific languages and how these interpreters implement incremental code updates.

[1] Because the Karel language does not have a primitive to turn right, we have to implement turning right by turning left three times.

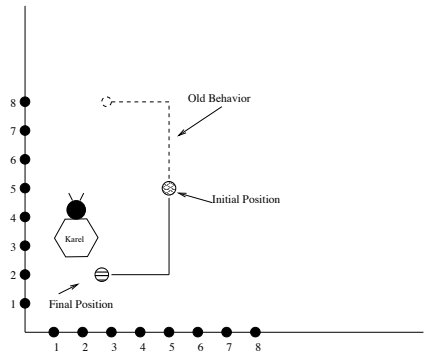

Figure 12.5: The incremental change to Karel's behaviour.

```
root().child(1,times).child(1,turnleft)
```

Figure 12.6: XPointer Address for Increment.

12.3 Implementation of the Approach

After a programming language has been specified, an interpreter for this language needs to be implemented. We first show how significantly off-the-shelf XML technology, most notably XML Parsers and the implementations of the Document Object Model, simplify the construction of such interpreters. Then, we explain how the communication between sender and receiver can be achieved using distributed object technology. Finally, we focus on the implementation of incremental code mobility, demonstrate how XPointer processors support locating the increment to be updated, and how the DOM supports incremental syntax tree modifications.

12.3.1 Interpreter Implementation

The first stage of an interpreting a program involves the validation of the syntactic correctness. As a result of that stage, interpreters produce an attributed abstract syntax tree (AST) of the program. If the program is written in XML, both tasks can be entirely delegated to a validating XML parser. We use IBM's XML4J [Alp99] but many other validating XML parsers exist. Figure 12.7 shows the use of the XML4J parser in our Java-based Karel interpreter. When invoking `parse` on the Karel code of Figure 12.2 the XML parser will construct the parse tree that is graphically represented in Figure 12.8.

```
import org.w3c.dom.*;          //DOM API
import com.ibm.xml.parser.*;  //XML Parser

public void execute(String program,
                    String update_location){
 ...
 //create a new parser for Karel Programs
 Parser parser=new Parser("Karel.dtd");
 InputStream is;
 // parser to read input stream from program
 is=new StringBufferInputStream(program);
 // root of parse tree for program in inc
 Document inc=parser.readStream(is);
 ...
}
```

Figure 12.7: Translating XML program into an AST.

The next stage of the interpretation is a static semantic analysis that checks, for example, the uniqueness of identifiers or the correct typing of expressions. This is often done while the interpreter is executing the code in order to avoid several traversals of the abstract syntax tree. Thus, while traversing the tree and visiting each node, the interpreter first checks for violations of the static semantics and then executes the operation that the node represents. Operations for traversals through ASTs that have been constructed from XML documents are standardized by the Document Object Model (DOM) [ABC+98] and are implemented in off-the-shelf products, such as IBM's XML4J. The DOM traversal operations support obtaining all the children of a node, querying the type of the node, obtaining values of node attributes and so on.

Figure 12.9 shows an excerpt of the Karel interpreter that traverses the abstract syntax tree and executes a statements for each AST node. The actions usually modify some state variables. In case of Karel, these state variables indicate whether the robot has been switched on, its current position and direction and the number of items that it has picked up. The interpretation is then performed as a recursive method `execute`, which is initially passed the root node of the AST tree. It then examines the type of node and performs the

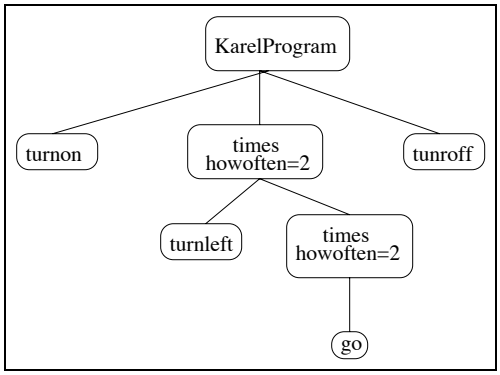

Figure 12.8: Abstract Syntax Tree for Karel's Program.

appropriate action. For the root node, it recursively calls execute for all its child nodes. For a node of type `go` , it adds the current direction to its co-ordinates. We note that for Karel, the implementation of each command requires a few lines of code and in total is about 50 lines of Java code.

12.3.2 Code Mobility

In order to support code mobility, we have to send an XML program from a remote host rather than read it from a local file system. Any transfer protocol could be used for this purpose. However, in [ESF99], we discussed the benefits of using distributed object technology to transport XML documents between different hosts of a network. The same considerations apply to XML documents that represent programs and we therefore use distributed object technology to pass XML programs from a sender that manages the execution to a receiver that then implements the interpreter as shown above. Figure 12.10 visualizes this behaviour.

For sending Karel programs to the robot interpreter, we use Java/RMI [RMI98]. The use of distributed object technology rather than lower-level network protocols is motivated by the availability of further middleware services. If for example security is important in a particular application area and use of the interpreter by non-authorized principals needs to be disabled or requesters need to be authenticated, a security service, such as the CORBA security service [OMG98a] could be used.

In order to facilitate the remote communication that transmits the mobile code, the Karel Interpreter declares the remote interface `Karel` as shown in Figure 12.11. That

```
import org.w3c.dom.*;        // DOM API
import com.ibm.xml.parser.*; // XML parser

class KarelExecutor {
 //the position and direction of Karel:
 private int x_pos=5, y_pos=5;
 private int x_direction=1, y_direction=0;
 private int num_beepers=0; //collected items
 private boolean on=false;//activation status

 public void execute(Node n) {
  if (n.getNodeName().equals("KarelProgram")){
   NodeList children=n.getChildNodes();
   Node command;
   for (int i=0; i<children.getLength();i++) {
     execute(children.item(i));
   }
  } else if (n.getNodeName().equals("go")){
   if (on) {
    x_pos=x_pos+x_direction;
    y_pos=y_pos+y_direction;
   }
  } else ...
 }
}
```

Figure 12.9: Traversing the AST during interpretation.

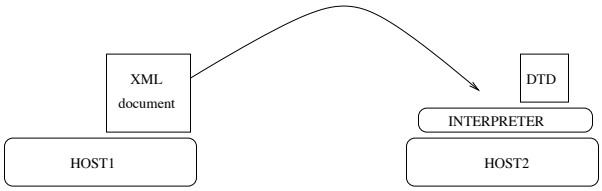

Figure 12.10: Migration of XML program to remote interpreter.

interface is implemented by the Karel interpreter. This enables a controller that resides

```
import java.rmi.*;
import java.io.*;

public interface Karel extends Remote {
 void execute(String program,
              String update_location)
      throws RemoteException,
             UnambiguousInsertException;
} // Karel
```

Figure 12.11: Remote Method Invocation for Karel.

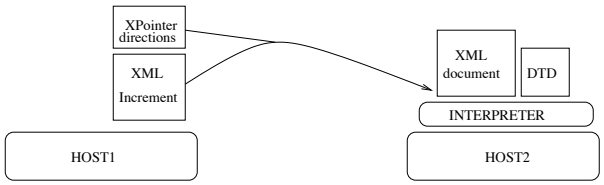

Figure 12.12: The migration of the increment XML file to the robot site.

on one host to send Karel programs for interpretation on a different host. Note that we do not transfer the DTD together with the code but rather assume that the DTD is stored locally. This choice derives from the observation that the interpreter implementation is very tightly linked to the DTD, because the DTD is the grammar of the language and every interpreter is dependent on the grammar of the language that it executes.

12.3.3 Incremental Code Mobility

Incremental and fine-grained mobility as shown in Figure 12.12 can be implemented using standard XML off-the-shelf technologies. So far, we have shown how to parse and interpret the program, which is passed as the first parameter to the `execute` method in Figure 12.11. The second parameter is an XPointer expression. If this XPointer expression is not empty and well-formed, it will identify a node in the abstract syntax tree that needs to be replaced with the program increment that is passed as the first parameter to `execute`. The strategy for implementing incremental code mobility is then as follows: we first parse the program increment passed as the first parameter and construct an syntax tree for the increment, we then evaluate the XPointer expression and then replace the node addressed

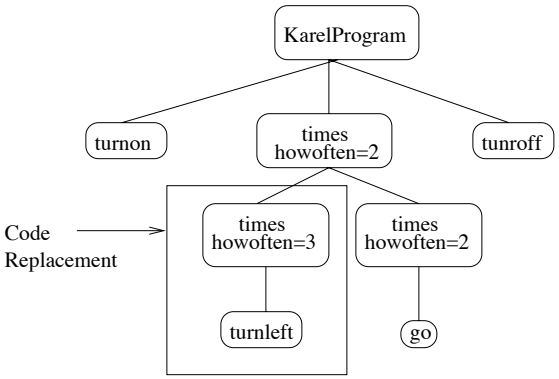

Figure 12.13: Result of incremental code update on AST.

in the expression with the root node of the syntax tree of the increment. This replacement is shown in Figure 12.13.

In order to implement this strategy for incremental updates, we take again advantage of the DOM. Parsing the program increment and constructing the AST for it is achieved in the same way as for the full program. This time, the parser just creates a tree whose root node type is different from the root type of the DTD. In case of our Karel increment, a root increment node of type `times` is created.

The evaluation of the XPointer expression for the replacement node can be fully delegated to an XPointer processor. Again there are several of those processors available and we use the one that comes with XML4J. Figure 12.14 shows how we use the XPointer processor in order to locate the node `replace` that needs to be replaced. The replacement of the code increment is shown at the bottom of Figure 12.14. We then navigate to the parent node of `replace` and substitute it with the root node of the syntax tree of the increment that was sent using standard DOM operations.

12.4 Applications

In the previous two sections, we have presented our work through a deliberately simple example in order to introduce our approach and highlight its potential. In this section, we describe application domains that could benefit from incremental code mobility with XML. These include user interfaces engine, the management of applications on portable digital assistants, and the flexible co-ordination of consistency checks in distributed documents.

```
import org.w3c.dom.*;        // DOM API
import com.ibm.xml.parser.*; // IBM's parser
import com.ibm.xml.xpointer.*;//IBM's xpointer
...
public void execute(String program,
                    String update_location)
       throws RemoteException,
              UnambiguousInsertException {
 ...
 // create an XPointer object from
 // the update location that is passed
 XPointerParser xpp=new XPointerParser();
 XPointer xp=null;
 xp=xpp.parse(update_location);
 // Interpret XPointer object from the
 //root node of the previously parsed doc
 Pointed nodelist=xp.point(root);
 if (nodelist.size()!=1) {
  throw new UnambiguousInsertException();
 } else {
  Node replace=(nodelist.item(0)).node;
  Node parent=replace.getParentNode();
  //we get the parent node
  if (parent==null)
   throw new UnambiguousInsertException();
  //replacement of the child with the new code
  parent.replaceChild(inc.getDocumentElement(),
                      replace);
 }
}
```

Figure 12.14: Evaluating XPointer Expression

12.4.1 User Interface Engines

The installation and administration of large-scale systems with thousands of clients is a potential application for incremental code mobility. The departure control system of

airlines that are used to handle check-ins are good examples. For large airlines or alliances, the clients implementing the user interface of such systems have to be deployed on several 10,000 machines, distributed across the globe. The machines are not necessarily owned by the airlines but are rather temporarily rented from airport authorities, which want to keep a tight regime on updates of software. Thus airlines cannot frequently update the software that is installed on these machines. To accommodate frequent changes, they have to utilize code mobility.

It would be possible to accommodate changes by deploying a Java Virtual Machine on each of these systems and downloading front-end applications from centralized servers. The Java approach, however, has two disadvantages. First it requires code of substantial size to be downloaded from a server, possibly through slow dial-up networks. Second, the Java code needs to be changed whenever the user interface needs to be changed. These limitations can be overcome by installing a general-purpose user interface engine onto each of the client machines that interpret high-level user interface descriptions.

XwingML is a DTD for such a user interface description language [Sof99b]. It provides markup tags for all Java Swing user interface components and also provides an interpreter for XwingML documents that generates the desired user interfaces. Applying our approach of code mobility to XML, the high-level descriptions of user interfaces can be sent from a centralized server to all distributed client hosts. Because the user interface descriptions are rather small compared to the size of the Java byte code of the full user interface application, we avoid the first of the above problems. The second limitation is overcome because the user interface description is just code, which can be generated by server applications that may, for example, be driven by business processes.

Incremental mobility can be applied successfully in this context, too. If the displayed window needs to be updated, for example by adding or replacing some buttons, an XML code increment can be sent to the user interface engine. The idea is exactly the same as with incremental code mobility for Karel the Robot. The program increment can be dynamically integrated with the original XML code for the window, thus making the window change its appearance.

12.4.2 Application Management on Mobile PDAs

An interesting application for our XML-based approach to code mobility arises when logical mobility meets physical mobility. Lightweight computing devices, such as Personal Digital Assistants (PDAs) are starting to be used for mission critical computing and are integrated into enterprise computing environments. In these settings, it is important for all PDAs to run the same set of applications. An example for such a PDA deployment is the New York Stock Exchange (NYSE). NYSE has equipped its traders with PDAs, that are used for trade data entry and automated transmission between trade data and back office trade settling systems.

The applications that are used in financial markets have to evolve rather rapidly. Financial analysts invent new products known as derivatives on a regular basis. Once such a product has been created, the trading applications need to be adjusted and be updated to support trading in these new derivatives. If the used machines were wired workstations it would be feasible to transfer and replace the complete code when needed. The incremental approach described in this chapter could be also applied. This approach becomes rather essential when the used devices are thin clients like PDAs; in this case incremental code updates are an interesting option, considering temporal unreachability of PDAs and slow IRDA or radio network connectivity.

To pursue this approach application developers have to devise an XML-based scripting language for developing trading applications. They also have to build an interpreter for this language which then is deployed on each PDA. Whenever an application needs to be adjusted, a program increment can be added to a list of updates that are kept on the server to which PDAs connect. Whenever a PDA physically enters the trading room and establishes connection to the server, the server first checks the patch-level of the applications on that PDA. The server will then incrementally send all application updates that are not yet deployed on the PDA.

The definition of an application-specific language and its implementation in an interpreter may sound difficult to accomplish. It is, however, well supported. The application-specific language can refer to XWingML for user interface definition purposes. The implementation of an interpreter is simplified by the availability of light-weight XML parsers and Java Virtual Machines that have already been developed for PDAs, such as 3COM's Palm Pilot [Sof99a].

```
<ConsistencyRule id="r1" type="CT">
 <id>r1</r1>
 <Description>
 For every instance in a collaboration diagram
 there must be a class in a class diagram
 with the same name.
 </Description>
 <Source>
  <XPointer>
   root().child(all,Package).
    (all,CollaborationDiagram).
    (all,Collaboration).(all,Instance)
  </XPointer>
 </Source>
 <Destination>
  <XPointer>
   root().child(all,Package).
    (all,ClassDiagram).(all,Class)
  </XPointer>
 </Destination>
 <Condition expsource="origin().attr(CLASS)"
  op="equal"
  expdest="origin().attr(NAME)"/>
</ConsistencyRule>
```

Figure 12.15: A Consistency Rule in XML Format.

12.4.3 Consistency Management

In [EEF+99], we describe a high-level language for defining rules that define the consistency between distributed software engineering documents. We assume that these documents are represented in XML themselves. This is a fair assumption, because Microsoft's Office 2000 can save documents in XML format, IBM's Visual Age environment uses XML as representation scheme for its project repository and most case tools can export UML diagrams in XMI.

[EEF+99] suggests a language to express consistency rules. This language is, in fact,

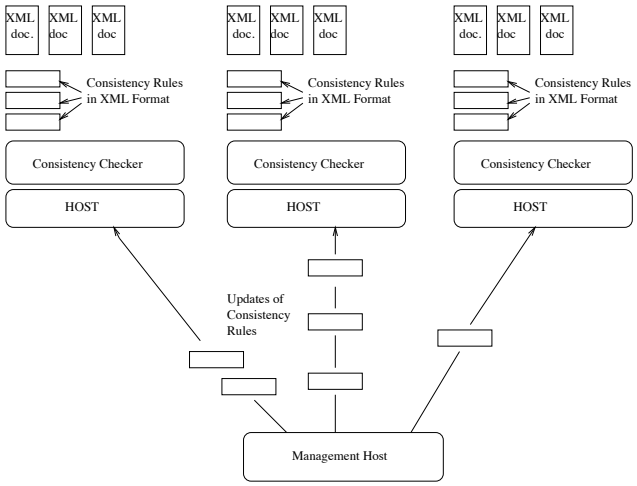

Figure 12.16: Consistency management architecture.

a high-level XML programming language and facilitates representing consistency rules as XML documents. Figure 12.15 shows an example of such a rule.

The rule is based on the XMI DTD and demands that for each object in each collaboration diagram, there is a class in a class diagram whose name equals the type of the object. In [EEF+99], we also explain the interpreter that executes these consistency rules in order to check the consistency of XML documents. The result of such a check for a rule is a set of XLink expressions that link consistent document fragments with each other and inconsistent document fragments to an indicator of such inconsistency.

The approach as described in [EEF+99] uses one set of rules and one rule interpreter. This is rather inflexible as every member of a software development team has to work against the same set of rules. Moreover, the centralized interpretation of rules creates a bottleneck that can be avoided if we have multiple rule interpreters on each developer's machine. The rule interpreter would then only have those sets of rules that the developer needs to check consistency of the documents she produced locally. We can even have dedicated interpreters for particular subgroups of the development team in order to check consistency between documents produced by different team members and then at a higher level there can be rule sets that check for project-wide consistency.

The sets of rules that are active at each of these interpreters cannot be static but have to evolve during the course of the project, for example as a result of changing team structures and different allocations of responsibilities. In order to accommodate such changes, the

set of rules that are active at each interpreter have to be changed. New rules have to be added and existing rules may have to be deleted. These changes can be triggered by a consistency supervisor component that uses our approach to incremental code mobility to pass the XML-encoded consistency rules to the different rule interpreters involved.

Figure 12.16 shows the overall architecture of this approach. Each developer's workstation and group and project servers run an interpreter for XML-consistency rules. The consistency supervisor manages the rule set of consistency rules that are active for each of these interpreters and moves new rules incrementally to these interpreters, if necessary.

12.5 Evaluation

In this section we discuss the advantages and current disadvantages of the approach. We also hint at how the disadvantages may be overcome.

We have demonstrated how XML and its related technologies can be used for both specifying and implementing incremental code mobility at any granularity. By not fixing a particular granularity for mobile code, we enable complete programs as well as individual lines of code to be sent across the network. The combination of fine-grained and incremental mobility achieves previously unavailable degrees of flexibility. We have examined the application of incremental and fine-grained code mobility to user interface management, application management on PDAs and consistency management of distributed documents.

The success of the approach critically depends on the ability to encode a high-level programming language in an XML DTD. Our Karel example has demonstrated that this is possible. The XwingML DTD suggests this can also be achieved in a scalable way. We can imagine, that our approach will be used to write XML versions of interpreted languages, such as Javascript. We could then build a compiler that translates between Javascript and the XML encoding and a XML interpreter that wraps an existing Javascript interpreter. Our approach then facilitates incremental code updates.

In the Karel example, we have only shown how incrementality can be achieved by replacing existing fragments. We note that this may be overly restrictive. However, the strategies shown here can also be applied to add or delete pieces of code to or from the original program. To address insertion points or identify the fragments of deletion, we

could use XPointer in the same way. To implement the changes to the abstract syntax tree, we could use the `insert` and `delete` method calls of the XML4J package [Alp99].

Our approach has not yet explored the combination of data and code mobility, in a step towards agent mobility. This would be, however, a rather small improvement as XML is naturally well suited to express data structures. To achieve this in our Karel example, we could change the DTD of Karel's language and add an encoding for the position and other state attributes of Karel. In this way we can write an XML program containing Karel's position initialization. The interpreter would have to be modified as well in order to be able to obtain the information (i.e. the initial position), and to initialize Karel's status correctly.

We used Java and RMI for implementing the migration of the XML program in the example, the approach, however, is completely language independent, as long as XML is used to encode the moving code[2].

The incremental update of the code is done after the robot has terminated an execution. However, in some applications it may be convenient to apply the changes to the program while the program is executing. The user interface application is a good example. This is feasible in our approach as well. Nevertheless, it would raise problems related to the maintenance of the program counter and the updating of operations in a cycle. However, if the language is simple enough this might be feasible.

Furthermore, incremental updating of code raises a series of issues related with access control problems: for instance, what happens if the code is updated twice by different principals? No one of the parties would know the actual status of the program. In our perspective we see applications in "code-distribution" oriented domains, where a single sender has full control of the code and has the right to update it. If we did not use RMI, but CORBA to transmit the code, the CORBA security service could be used to enforce these access rights.

[2]The use of Java was also driven by the availability of XML4J [Alp99] tools.

Chapter 13

Summary

In Chapter 11 we have presented a tool for visualizing Z specifications on the WWW: it fits every browser and every platform. The tool is based on XML displayed through "displets". The advantage of having a Z browser running on all platforms is essentially that sharing of Z documents is encouraged by the diffusion of WWW on the Internet.

A possible application can be a groupware tool for editing and versioning formal documents; such a tool could be integrated with other software tools in order to improve the specification phase of the software process. The reuse of parts of documents obviously benefits from having these hypertextual Z documents. The tools will also improve the search of pieces of specifications in complex documents: every element in the Z specification can be labeled or linked to other pieces of documents or to URL on the Internet.

XML can be further extended in order to include new symbols and integrate Z specification with other notations: new Java classes have to be written for the new symbols.

An ambitious goal consists of defining all XML displets necessary in an organization to support the intranet management system of formal documents typical of such an organization. For instance, we are currently working on displets for managing UML documents. XML documents are completed with XSL style sheets that instruct display engines about how the rendering of notations. This first attempt toward the use of XML technologies for code mobility is enhanced in Chapter 12, where we showed a way of applying the fine-grained code mobility to specific application domains. We used off-the-shelf tools to implement a very fine-grained approach and used it in different contexts, validating the thesis that fine-grained code mobility could be actually be useful in specific studies, and opening the doors to new approaches.

In the chapter we presented an incremental approach to code mobility using the XML language. The novelty of the approach consists in being able to send code incrementally instead of re-sending complete updated versions of the code. Java based technologies launched the idea of object and classes mobility, allowing a set of new paradigms for communication to become feasible [WPM99, Pic98].

Many theoretical languages have been used to specify and analyze code mobility [CG98, MR98, NFP98, CFM98, FGL+96] the movement is specified with different granularities showing that the Java point of view, where a class is the unit of mobility was not the only possibility to be explored [MPR99]. In this part we showed a possible incarnation of these ideas, and described a set of application in a domain specific contexts. Possible developments of this work may involve security issues: incrementally updating of code raises access right and authentication issues.

Incremental update of executing code, although already feasible in the approach, is a challenging field we will explore, maybe restricting to specific its use to particular domains. We intend to explore the use of this approach in real projects involving industrial partners in some of the domains that we mentioned in Section 12.4.

Conclusions

The main goal of the work presented in this thesis has been the development of models, tools, and prototypes allowing the investigation and the exploitation of mobile code techniques. More specifically, we concentrated on mobile code specification and automatic analysis, on investigation of basic mobile code operations and exploiting the power of logical mobility looking beyond the developed mobile code technologies. Some of the results of the investigation could be applied to application domains through the use of existing technologies in a way that was compliant with our investigation.

Contribution

We now summarize the contribution of this thesis.

Specification and Analysis of Mobile Code System

We have presented the specification language PoliS and its enhanced version (MobiS) able to specify mobile code features of systems. On top of PoliS a model checker has been built by our group in Bologna for the analysis of properties of systems. In this thesis we used the model checker to prove properties on mobile code systems. MobiS allows the specification of mobile agents as first class elements nd the specification of three major mobility paradigms,i.e., data, code and agent mobility can be formalized.

Model and Prototype for Fine-grained Mobility

Mobile UNITY has been used to give semantics to a set of basic primitives for logical mobility and for investigating issues related with fin-grained mobility. In this approach every line of code and every variable is considered mobile. In the model we study the details of the approach trying to constrain it to follow existing technology behavior. The

prototype (Lilliput) presented was written in Java . It shows the implementability of the semantics constrains and of the basic operations of the model.

Implementation of Incremental Code Mobility

In the last part of the thesis we have presented the use of currently available techniques and tools for the implementation of a very fine-grained model for code mobility. The approach uses XML and related technologies for achieving incremental mobility at a statement level. The number of possible application domains for this approach reveals that the existing technology might sometimes be inadequate. We showed some possible applications of this new approach.

Future Work

Different lines of future work could be followed. I will distinguish future work on the different parts in order to show these multiple lines. On the specification and analysis front with PoliS and MobiS much can be done in order to improve the readability of the specifications. We are thinking of possible integration of PoliS with UML and XML in order to obtain a more user-friendly interface. The model checker needs to be enhanced in order to deal with MobiS specifications. We are investigating this issue as having agents as first class elements leads to a large state explosion in our current model checker. Security properties could result to be very interesting on an analysis front, especially in a code mobility setting.

On the fine-grained approach we are interested in enhancing some aspects such as referencing units by type and not by name, and exploring other mobility primitives not considered. In Chapter 8 we enhanced the model with nested spaces and it would be interesting to study referencing in mode detail in the tree structure generated by nested processes. A prototype of the nested model could be implemented as well exploring implementation problems of the approach. Security issues could be explored on this fine-grained approach as well, especially the ones concerned with resource access and remote entity referencing and migration.

The use of Mobile UNITY temporal logic proof system would offer a good field for analysis of properties on the model with respect to incremental code mobility with XML.

We see future work in the application domains described in Chapter 12. We also have plan for application of the XML approach to mobile computing settings and Java cards. XML technologies are rapidly evolving and new features can be introduced in the prototype like for instance XML schemas could substitute XML DTDs allowing a higher level of flexibility.

Closing Remark

Analysis, specification, and prototyping have revealed to be powerful instruments for investigation of new paradigms and technologies. With respect to mobile code the formal approach taken led us to the development of challenging new models, independent from the developed mobile code technologies. From model we went down again reaching the technology level and demonstrated the implementability of the new approaches that revealed potentially unexplored trends for code mobility in different application domains.

Appendix A

The Lilliput Input Grammar

S	→	**System** *Id SystemBody* **end**
SystemBody	→	*Programs Components*
Programs	→	*Id ProgramBody* **end**
Components	→	**Components** *ComponentsBody* **end**
ProgramBody	→	*OptDeclare OptInitialize OptAssign*
OptDeclare	→	**declare** *Declare*|ϵ
Declare	→	*Declaration*
	|	*Declaration* [] *Declare*
Declaration	→	*Id* **:** *DeclarationBody*
DeclarationBody	→	**var** *Integer*
	|	*Integer*
OptIntialize	→	**initially** *Initialize*|ϵ
Initialize	→	*Initialization*
	|	*Initialization* [] *Initialize*
Initialization	→	*Id* = *RightInit*
RightInit	→	*Integer*
	|	*Function*(*ListId*)
OptAssign	→	**assign** *Assign* **end**|ϵ

Assign	→	*Assignment*
	|	*Assignment* [] *Assign*
Assignment	→	*Label* **:** *Stmt*
Stmt	→	*St* || *Stmt*
	|	*St*
St	→	*GuardedStmt*
	|	*SimpleStmt*
	|	*QuantifiedStmt*
GuardedStmt	→	[*Stmt*] **if** (*Guard*)
SimpleStmt	→	*Id* := *Rightside*
Rightside	→	*Integer*
	|	*Function*(*ListId*)
	|	*MobilityCalls*
QuantifiedStmt	→	< *OpListId* **:** *Range* :: *Stmt* >
Range	→	*Limit LRel Id RRel Limit*
LRel	→	$< \mid \leq$
RRel	→	$> \mid \geq$
Limit	→	*Integer*|*Id*
Guard	→	*SimpleGuard Relop Guard*
	|	*SimpleGuard*
SimpleGuard	→	*Term*
	|	*SimpleGuard Addop Term*
Term	→	*Factor*
	|	*Term Mulop Factor*
Factor	→	*Id*
	|	*Function*(*ListId*)
	|	*Integer*
	|	**not** *Factor*

$$
\begin{array}{lcl}
\mathit{Op} & \rightarrow & [] \mid \| \\
\mathit{MobilityCalls} & \rightarrow & \mathit{Predicate}(\mathit{List}) \\
\mathit{Predicate} & \rightarrow & \textbf{move}|\textbf{put}|\textbf{clone}|\ldots \\
\mathit{List} & \rightarrow & \mathit{Id},\ \mathit{List}|\mathit{Function}(\mathit{List}),\ \mathit{List} \\
 & | & \mathit{Id}|\mathit{Function}(\mathit{List}) \\
\mathit{ComponentsBody} & \rightarrow & \mathit{Component}\ []\ \mathit{ComponentsBody} \\
\mathit{Component} & \rightarrow & \mathit{QuantifiedC}\ []\ \mathit{SimpleC} \\
\mathit{SimpleC} & \rightarrow & \textbf{newData}(\mathit{List}) \\
 & | & \textbf{newCode}(\mathit{List}) \\
 & | & \textbf{newProcess}(\mathit{List}) \\
\mathit{QuantifiedC} & \rightarrow & <\ []\ \mathit{Id} : \mathit{Range} :: \mathit{SimpleC} > \\
\mathit{RelOp} & \rightarrow & \leq \mid \geq \mid < \mid > \\
\mathit{Addop} & \rightarrow & + \mid - \mid \vee \\
\mathit{mulop} & \rightarrow & * \mid / \mid \wedge \\
\mathit{Id} & \rightarrow & \mathit{letter}\ (\mathit{letter}|\mathit{digit})^* \\
\mathit{Integer} & \rightarrow & \mathit{digits}\ \mathit{optional} - \mathit{exp} \\
\mathit{digits} & \rightarrow & \mathit{digit}\ \mathit{digit}^* \\
\mathit{optional} - \mathit{exp} & \rightarrow & (E(+ \mid - \mid \epsilon)\ \mathit{digits}) \mid \epsilon
\end{array}
$$

Appendix B

The Lilliput API

Hierarchy For Package lilliput

- class java.lang.Object
  - class lilliput.**LilliElement** (implements java.lang.Cloneable, lilliput.LilliConstants, java.io.Serializable)
    - class lilliput.**LilliProcess**
    - class lilliput.**LilliUnit**
      - class lilliput.**LilliCU**
      - class lilliput.**LilliDU**
  - class lilliput.**LilliEngine** (implements lilliput.LilliConstants)
  - class lilliput.**LilliHandler** (implements mucode.GroupHandler, lilliput.LilliConstants)
  - class java.lang.Thread (implements java.lang.Runnable)
    - class lilliput.**LilliInterpreter** (implements lilliput.LilliConstants)
  - class java.lang.Throwable (implements java.io.Serializable)
    - class java.lang.Exception
      - class lilliput.**LilliActivateException**
      - class lilliput.**LilliDestroyException**
      - class lilliput.**LilliExecutableException**
      - class lilliput.**LilliFindException**
      - class lilliput.**LilliMoveProcessException**

        - class lilliput.**LilliReferenceException**

    - class lilliput.**Variable** (implements java.lang.Cloneable, lilliput.LilliConstants, java.io.Serializable)

- interface lilliput.**LilliConstants**

Interface LilliConstants

All Known Implementing Classes: LilliInterpreter, LilliElement, LilliHandler, LilliEngine, Variable

---

public interface LilliConstants

Constants for the package LilliConstants.java

Field Summary

`static int` **ACTIVE** status of an active process

`static int` **CODE** type of code units

`static int` **DATA** type of data units

`static java.lang.String` **EMPTY** empty string

`static int` **INACTIVE** status of an inactive process

`static int` **PROCESS** type of processes

`static int` **TERMINATED** status of a terminated process

`static int` **UNDEFINED** value assigned to variables that do not need to carry a value

---

Class LilliElement

```
java.lang.Object
  |
  +--lilliput.LilliElement
```

Direct Known Subclasses: LilliProcess, LilliUnit

---

public abstract class LilliElement

extends java.lang.Object

implements java.io.Serializable, java.lang.Cloneable, LilliConstants

The class defines the abstract element LilliElement.java

Field Summary

`java.lang.String` **name** the name of the element

`java.lang.String` **program** the name of the program the element comes form in case of a process the program is the program where the contained units come from

`int` **type** the type of the element

Fields inherited from interface lilliput.LilliConstants

`ACTIVE, CODE, DATA, EMPTY, INACTIVE, PROCESS, TERMINATED, UNDEFINED`

Constructor Summary

LilliElement`()`

Class LilliUnit

```
java.lang.Object
  |
  +--lilliput.LilliElement
        |
        +--lilliput.LilliUnit
```

Direct Known Subclasses: LilliCU, LilliDU

---

public abstract class LilliUnit

extends LilliElement

defines the class of a general (abstract) unit LilliUnit.java

Field Summary

`LilliProcess` **container** name of the containing process

`java.lang.String` **destination** the name of the destination process when the unit is to be moved

`java.util.LinkedList` **referencedBy** the list of the referencing processes

Fields inherited from class lilliput.LilliElement

`name, program, type`

Constructor Summary

LilliUnit`()`

Method Summary

`java.lang.Object` **clone**`()` the method to clone the unit

Class LilliProcess

```
java.lang.Object
  |
  +--lilliput.LilliElement
        |
        +--lilliput.LilliProcess
```

---

public class LilliProcess

extends LilliElement

See Also: Serialized Form

Field Summary

`java.util.LinkedList` **codeList** the list of contained code units

`java.util.LinkedList` **dataList** the list of contained data units

`java.util.LinkedList` **gamma** the list of referenced units

`mucode.MuServer` **muserver** the muserver proper of the process where to store the classes of the units

`int` **status** the staus of a process: the initalization value is to INACTIVE

Fields inherited from class lilliput.LilliElement

`name, program, type`

Constructor Summary **LilliProcess**`(java.lang.String n, java.lang.String prog)` the constructor with two parameters

Method Summary `java.lang.Object` **clone**`()`

the method clones the process

`java.lang.Object` **cloneUndef()** the cloning of a process with initialization values

---

Class LilliDU

```
java.lang.Object
  |
  +--lilliput.LilliElement
        |
        +--lilliput.LilliUnit
              |
              +--lilliput.LilliDU
```

---

public class LilliDU

extends LilliUnit

contains the class for a data unit LilliDU.java

Field Summary

`Variable` **var** the variable contained in the unit

Fields inherited from class lilliput.LilliUnit

`container, destination, referencedBy`

Fields inherited from class lilliput.LilliElement `name, program, type`

Constructor Summary

LilliDU`(java.lang.String n, java.lang.String p, int val)` the method clones the data unit

Method Summary

`java.lang.Object` **clone()** the method to clone the unit

`java.lang.Object` **cloneUndef()** the method clones the unit with undefined value for the variable; the method needs to be refined by the real data unit class that can initialize the variable with the actual initialization value

Class LilliCU

```
java.lang.Object
  |
  +--lilliput.LilliElement
        |
        +--lilliput.LilliUnit
              |
              +--lilliput.LilliCU
```

---

public class LilliCU

extends LilliUnit

This is the class representing a code unit LilliCU.java

Field Summary

`java.util.LinkedList` **vars** the list of variables of the unit

Fields inherited from class lilliput.LilliUnit

`container, destination, referencedBy`

Fields inherited from class lilliput.LilliElement

`name, program, type`

Constructor Summary

LilliCU`(java.lang.String n, java.lang.String p)` the method containing the actual code: to be overwritten

Method Summary

`java.lang.Object` **clone**`()` the method to clone che code unit.

`java.lang.Object` **cloneUndef**`()` the method clones the unit but leaves all the variables to undefined `void` **perform**`()` method containing the code of the unit

---

Class LilliEngine

```
java.lang.Object
  |
  +--lilliput.LilliEngine
```

---

public class LilliEngine

extends java.lang.Object

implements LilliConstants

LilliEngine.java

Field Summary `(package private) static LilliEngine` **engine** the actual engine reference

`(package private) java.util.LinkedList` **executable** the list of executable units

`(package private) static LilliInterpreter` **interpreter** the static interpreter

`(package private) java.util.LinkedList` **libraryCU** the list of code units part of the host library

`(package private) java.util.LinkedList` **libraryDU** the list of data units part of the host library

`(package private) mucode.MuServer` **muserver** the muserver storing the classes on the host

`private java.util.LinkedList` **pending** the list of pending elements to enter the host pending is accessed only by two synchronized methods add and removePending

`(package private) java.util.LinkedList` **pList** the list of processes on the host

Fields inherited from interface lilliput.LilliConstants

`ACTIVE, CODE, DATA, EMPTY, INACTIVE, PROCESS, TERMINATED, UNDEFINED`

Constructor Summary

LilliEngine`()`

Method Summary

`void` **activate**`(LilliProcess p)` 'activate' method it activates an inactive process.

`(package private) void` **addPending**`(LilliElement el)`

`LilliElement` **cloning**`(LilliElement e)`

`void` **deactivate**`(LilliProcess p)` 'deactivate' method to deactivate a process so that it does not execute

`void` **destroy**`(LilliElement e)` 'destroy':method to destroy an element.

`private void` **eliminate**`(LilliElement e, LilliProcess father)` 'eliminate' method: to eliminate an element from the site.

`LilliElement` **find**`(java.lang.String s)`

`static LilliEngine` **getEngine**`()`

`static void` **main**`(java.lang.String[] args)`

`void` **move**`(LilliElement e, java.lang.String lambda, java.lang.String ProcessName)` the method move moves an element to a location the movement can address also a destination process if the element to be moved is a unit.

`void` **newCode**`(LilliCU c, java.lang.String lambda)`

`void` **newData**`(LilliDU d, java.lang.String lambda, int val)`

`void` **newProcess**`(LilliProcess p, java.lang.String prog, java.lang.String lambda, int status)`

`LilliElement` **neww**`(java.lang.Class c)`

`void` **parseArgs**`(java.lang.String[] args, int index)`

`private static void` **printHelp**`()`

`LilliCU` **put**`(LilliCU c)`

`LilliDU` **put**`(LilliDU d)`

`LilliProcess` **put**`(LilliProcess p)`

`void` **reference**`(LilliProcess p, LilliUnit u)` to reference a local element not of type process

`(package private) LilliElement` **removePending**`()`

`private void` **searchAndRemove**`(java.util.LinkedList l, LilliElement e)` auxiliary method to search and remove from a list of elements

`void` **terminate**`(LilliProcess p)` to terminate a process

`void` **unreference**`(LilliProcess p, LilliUnit u)` to unreference an element.

Class LilliHandler

```
java.lang.Object
  |
  +--lilliput.LilliHandler
```

---

public class LilliHandler

extends java.lang.Object

implements mucode.GroupHandler, LilliConstants

The handler called by the MuCode server. LilliHandler.java

Field Summary

`(package private) LilliEngine` **actualEngine**

Fields inherited from interface lilliput.LilliConstants

`ACTIVE, CODE, DATA, EMPTY, INACTIVE, PROCESS, TERMINATED, UNDEFINED`

Constructor Summary

LilliHandler`()`

Method Summary

`java.lang.Runnable` **unpack**`(mucode.Group group, mucode.MuServer server)` the method called by mucode to unpack the group arrived at destination

---

Class LilliInterpreter

```
java.lang.Object
  |
  +--java.lang.Thread
        |
        +--lilliput.LilliInterpreter
```

---

public class LilliInterpreter

extends java.lang.Thread

implements LilliConstants

The class of the interpreter LilliInterpreter.java

Field Summary

`(package private) LilliEngine` **actualEngine**

Fields inherited from interface lilliput.LilliConstants

`ACTIVE, CODE, DATA, EMPTY, INACTIVE, PROCESS, TERMINATED, UNDEFINED`

Constructor Summary

LilliInterpreter(LilliEngine en)

Method Summary

private void **bind**() bind(): to bind the variables of the units on the host

private void **bindDUCU**(LilliProcess p, LilliDU d) this method binds a data unit d to the code unit in the same scope

private void **bindDUDU**(LilliProcess p, LilliDU d) this method binds a data unit d to the data unit in the same scope

private void **bindDUGamma**(LilliProcess p, LilliDU d) this method binds the data unit d to the referenced units in the same scope

private void **enable**() this method enables the code unit ready for execution and put them in the executable list of the engine

private void **engage**() 'engage' method: to get from the pending list the elements to be arranged on the host, bind the variables and enable code units

void **eval**() this method picks up a code unit for execution and executes it

private void **execute**(LilliCU c) this method executes the unit chosen calling the method perform of the unit

private LilliElement **lilliFind**(java.lang.String s, java.util.LinkedList l) to find an element with name s in a list.

private void **linkDUvars**(Variable v1, Variable v2) this method binds two variable with same name

private void **merge**()

to get incoming units and put them on the right place on the host

private void **moveClassElement**(LilliElement e, mucode.MuServer muserver) this method helps in moving classes to a classpace

private LilliCU **pickUp**() this method picks up a unit for execution from the executable list with a random policy

void **run**() the run method executes the engagement and the evaluation

---

Class Variable

java.lang.Object

|

```
+--lilliput.Variable
```

---

public class Variable

extends java.lang.Object

implements java.io.Serializable, LilliConstants, java.lang.Cloneable

the class defining a variable Variable.java

Field Summary

`java.lang.String` **name** the name of the variable

`int` **value** the value of the variable

Fields inherited from interface lilliput.LilliConstants

`ACTIVE, CODE, DATA, EMPTY, INACTIVE, PROCESS, TERMINATED, UNDEFINED`

Constructor Summary

Variable`(java.lang.String name1)` the constructor initializing the value to the undefined value

Variable`(java.lang.String name1, int value1)` the constructor for specific initialization with value

Method Summary `java.lang.Object` **clone**`()` to clone the variable

References

[ABC+98] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification. W3C Recommendation http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001, World Wide Web Consortium, October 1998.

[AG97] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, 6(3):213–249, June 1997.

[Alp99] IBM Alphaworks. XML4J. http://www.alphaworks.ibm.com/tech/xml4j, 1999.

[Ama97] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, volume 1282 of *LNCS*. Springer, 1997.

[Ban97] J. Bannan. *Intranet Document Management*. Addison-Wesley, 1997.

[BB92] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[BC98] J. Bowen and D. Chippington. Z on the Web using Java. In J. Bowen, A. Fett, and M. Hinchey, editors, *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, volume 1493 of *LNCS*, pages 66–80, Berlin, September 1998. Springer.

[BCM+92] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *Information and Computation*, 98(2):142–170, June 1992.

[BHL98] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. World Wide Web Consortium Working Draft. `http://www.w3.org/TR/WD-xml-names`, Sept 16, 1998.

[BJR99] G. Booch, I. Jacobson, and J. Rumbaugh. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[BL96] J. P. Banâtre and D. LeMétayer. Gamma and the Chemical Reaction Model: Ten Years After. In J.M. Andreoli, C. Hankin, and D. LeMétayer, editors, *Coordination Programming: Mechanisms, Models and Semantics*, pages 3–41. Imperial College Press, 1996.

[BLLJ98] B. Bos, H. Lie, C. Lilley, and I. Jacobs. Cascading Style Sheets, level 2 CSS2 Specification. W3C Recommendation. `http://www.w3.org/TR/REC-CSS2`, May 12, 1998.

[BN92] S. Brien and J. Nicholls. Z Base Standard, November 1992.

[BPSM97] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML). *The World Wide Web Journal*, 2(4):29–66, 1997.

[BPSM98a] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[BPSM98b] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language. Recommendation http://www.w3.org/TR/1998/REC-xml-19980210, World Wide Web Consortium, March 1998.

[Bro96] M. Broy. Experiences with Software Specification and Verification using LP, The Larch Prover Assistant. *Formal Methods in System Design*, 8(3):221–272, 1996.

[Car95] L. Cardelli. A language with distributed scope. In *Proc. 22nd ACM Symp. on Principles of Programming Languages (POPL)*, 1995.

[CCM96] P. Ciancarini, S. Cimato, and C. Mascolo. Engineering Formal Requirements: Analysis and Testing. In *Proc. 8th Int. Conf. on Sw. Eng. and Knowledge Eng. (SEKE)*, pages 385–392, Lake Tahoe, Ne, June 1996.

[CCM97] P. Ciancarini, S. Cimato, and C. Mascolo. Engineering Formal Requirements: an Analysis and Testing Method for Z Documents. *Annals of Software Engineering*, 3:189–220, 1997.

[CD98] J. Clark and S. Deach. Extensible Stylesheet Language (XSL), Version 1.0. World Wide Web Consortium Working Draft. `http://www.w3.org/TR/WD-xsl`, Aug 18, 1998.

[CES86] E. Clarke, E. Emerson, and A. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[CF99] S. Conchon and F. Le Fessant. Jocaml: Mobile Agents for Objective-Caml. In D. Milojicic, editor, *Proc. 3nd Int. Symp. on Mobile Agents*, Palm Springs, CA, October 1999. IEEE Computer Society Press.

[CFM98] P. Ciancarini, F. Franzè, and C. Mascolo. A Coordination Model to Specify Systems including Mobile Agents. In *Proc. 9th IEEE Int. Workshop on Software Specification and Design (IWSSD)*, pages 96–105, Japan, 1998.

[CFM00] P. Ciancarini, F. Franzè, and C. Mascolo. Using a Coordination Language to Specify and Analyze Systems containing Mobile Components. *ACM Trans. on Software Engineering and Methodology*, page To appear, 2000.

[CFR97] P. Ciancarini, A. Fantini, and D. Rossi. A multi-agent process centered environment integrated with the WWW. In *Proc. 6th IEEE Workshops on Enablings Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 113–120, Boston, June 1997. IEEE Computer Society Press.

[CG92] N. Carriero and D. Gelernter. Coordination Languages and Their Significance. *Communications of the ACM*, 35(2):97–107, February 1992.

[CG98] L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, volume 1378 of *LNCS*, Lisbon, Portugal, 1998. Springer.

[CG00] L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000.

[CGL94] E. Clarke, D. Grumberg, and D. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.

[CK96] S. Cheung and J. Kramer. Context Constraints for Compositional Reachability Analysis. *ACM Trans. on Software Engineering and Methodology*, 5(4):334–377, October 1996.

[CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[CM96] P. Ciancarini and C. Mascolo. Analyzing the dynamics of a Z specification. In J. Calmet and C. Limongelli, editors, *Proc. Int. Symp. on Design and Implementation of Symbolic Computation Systems (DISCO)*, volume 1128 of *LNCS*, pages 138–149, Karlsruhe, Germany, September 1996. Springer.

[CM97] P. Ciancarini and C. Mascolo. Analyzing and Refining an Architectural Style. In J. Bowen, M. Hinchey, and D. Till, editors, *Proc. 10th Int. Conf. on the Z Formal Method (ZUM)*, volume 1212 of *LNCS*, pages 349–368, Reading, UK, April 1997. Springer.

[CM98a] P. Ciancarini and C. Mascolo. Model checking a software architecture. In *Proc. of the Workshop on Software Architecture Analysis and Testing (ROSATEA98)*, June 1998.

[CM98b] P. Ciancarini and C. Mascolo. Software architecture and mobility. In D. Perry and J. Magee, editors, *Proc. 3rd Int. Software Architecture Workshop (ISAW-3)*, ACM SIGSOFT Software Engineering Notes, pages 21–24, Orlando, FL, November 1998.

[CM98c] P. Ciancarini and C. Mascolo. Using a Coordination Language to Specify the Invoicing System. In M. Allemand, C. Attiogbe, and H. Habrias, editors, *Proc. Int. Workshop on Comparing Systems Specification Techniques*, pages 67–82, Nantes, France, March 1998.

[CM98d] P. Ciancarini and C. Mascolo. Using Formal Methods to Teach Software Engineering: a Tool-based Approach. *Annals of Software Engineering*, 6:433–453, 1998.

[CM99] P. Ciancarini and C. Mascolo. Specification and analysis of component based software architectures. Proc. First IFIP Int. Working Conf. on Software Architecture, February 1999.

[CMP98] P. Ciancarini, M. Mazza, and L. Pazzaglia. A Logic for a Coordination Model with Multiple Spaces. *Science of Computer Programming*, 31(2/3):231–262, July 1998.

[CMV98] P. Ciancarini, C. Mascolo, and F. Vitali. Visualizing Z Notation in HTML Documents. In J. Bowen, A. Fett, and M. Hinchey, editors, *Proc. 11th Int. Conf. on the Z Formal Method (ZUM)*, volume 1493 of *LNCS*, pages 81–95, Berlin, September 1998. Springer.

[CRV98] P. Ciancarini, A. Rizzi, and F. Vitali. An extensible rendering engine for XML and HTML. *Computer Networks and ISDN Systems*, 30(1-7):225–238, 1998.

[CVM99] P. Ciancarini, F. Vitali, and C. Mascolo. Managing complex documents over the WWW: a case study for XML. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):629–638, July/August 1999.

[EEF+99] E. Ellmer, W. Emmerich, A. Finkelstein, D. Smolko, and A. Zisman. Consistency Management of Distributed Documents using XML and Related Technologies. Technical report, University College London, Dept. of Computer Science, 1999. Submitted for Publication.

[EFT92] R. Enders, T. Filkorn, and D. Taubner. Generating BDDs for symbolic model checking in CCS. In *Proceedings of Computer Aided Verification (CAV '91)*, volume 575 of *LNCS*, pages 203–213, Berlin, Germany, July 1992. Springer.

[EMF00] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing Incremental Code Migration with XML. In M. Jazayeri and A. Wolf, editors, *Proc. 22nd Int. Conf. on Software Engineering (ICSE2000)*, pages 397–406, Limerick, Ireland, June 2000. ACM Press.

[ESF99] W. Emmerich, W. Schwarz, and A. Finkelstein. Markup Meets Middleware. In *Proc. of the 7th Int. Workshop on Future Trends in Distributed Systems, Capetown, South Africa*. IEEE Computer Society Press, 1999. To appear.

[Eva94] A. Evans. Specifying and Verifying Concurrent Systems Using Z. In M. Bertran, T. Denvir, and M. Naftalin, editors, *Proc. FME'94 Industrial Benefits of Formal Methods*, volume 873 of *LNCS*, pages 366–380. Springer, 1994.

[FFFv97] M. Feather, S. Fickas, A. Finkelstein, and A. vanLamsweerde. Requirements and Specification Exemplars. *Automated Software Engineering*, 4(4):419–438, 1997.

[FGL+96] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*. Springer, 1996.

[FKV94] M. Fraser, K. Kumar, and V. Vaishnavi. Strategies for Incorporating Formal Specifications in Software Development. *Communications of the ACM*, 37(10):74–86, October 1994.

[FPV98] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[Fru98] S. Frunfrocken. Transparent Migration of Java-Based Mobile Agents. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 26–37, Stuttgart, Germany, 1998. Springer.

[GAO94] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In D. Wile, editor, *Proc. 2nd ACM SIGSOFT Symp. on Foundations of Software Engineering*, volume 19:5 of *ACM SIGSOFT Software Engineering Notes*, pages 175–188, New Orleans, USA, December 1994.

[GC95] D. German and D. Cowan. Experiments with the Z Interchange Format and SGML. In J. Bowen and M. Hinchey, editors, *Proc. 9th Int. Conf. on the Z Formal Specification Notation (ZUM)*, volume 967 of *LNCS*, pages 224–233, Limerick, Ireland, September 1995. Springer.

[GKC99] D. Giannakopoulou, J. Kramer, and S. C. Cheung. Analysing the behaviour of distributed systems using tracta. *Journal of Automated Software Engineering*, 6(1):7–35, 1999.

[Gra95] R. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, 1995.

[Gri97] R. Grimes. *DCOM Programming*. Wrox, 1997.

[GS90] S. Graf and B. Steffen. Compositional minimization of finite state systems. In *Proc. 2nd Int. Conf. on Computer-Aided Verification (CAV'90)*, New York, 1990. ACM Press.

[IM98] P. Ion and R. Miner. Mathematical Markup Language (MathML) 1.0 Specification. W3C Recommendation. `http://www.w3.org/TR/REC-MathML`, Apr 7, 1998.

[ISO86] ISO 8879. Information processing – Text and Office Systems – Standardised General Markup Language SGML. Technical report, International Standards Organisation, 1986.

[IW95] P. Inverardi and A. Wolf. Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Trans. on Software Engineering*, 21(4):373–386, April 1995.

[Jia94] X. Jia. *ZTC: A Type Checker for Z – User's Guide*. Institute for Software Engineering, Department of Computer Science and Information Systems, DePaul University, Chicago, IL 60604, USA, 1994.

[Jor91] D. Jordan. CADiZ - Computer Aided Design in Z. In S. Prehn and W. Toetenel, editors, *VDM 91: Formal Software Development Methods*, volume 551 of *LNCS*, pages 685–690. Springer, October 1991.

[Jum] Jumbo browser for CML. `http://www.venus.co.uk/omf/cml/`.

[KDJY97] G. Kaiser, S. Dossick, W. Jiang, and J. Yang. An Architecture for WWW-based Hypercode Environments. In *Proc. 19th Int. Conf. on Software Engineering (ICSE)*, pages 3–13, Boston, MA, May 1997.

[Knu68] D. E. Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[Knu84] D. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, May 1984.

[KZ97] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), 1997.

[L+95] D. Luckham et al. Specification and Analysis of System Architecture using RAPIDE. *IEEE Trans. on Software Engineering*, 21(4):336–355, April 1995.

[Lam86] L. Lamport. *LaTeX. User's Guide & Reference Manual.* Addison-Wesley, 1986.

[Lam94] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[LB97] H. Lie and B. Bos. *Cascading Style Sheets: Designing for the Web.* Addison-Wesley, 1997.

[LHM88] J. Levy, H. Hutchinson, and D. Moore. Fine-grained mobility in the Emerald System. *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[LM99] D. Lange and D. Milojicic, editors. *1st Int. Symp. on Agent Systems and Applications and 3nd Int. Workshop on Mobile Agents*, Palm Springs, CA, October 1999. IEEE Computer Society Press.

[LO98] D. B. Lange and M. Oshima. *Programming and Deploying Java™ Mobile Agents with Aglets™*. Addison-Wesley, 1998.

[M+95] L. Mikusiak et al. Z Browser: A Tool for Visualization of Z Specifications. In J. Bowen and M. Hinchey, editors, *Proc. 9th Int. Conf. on the Z Formal*

Specification Notation (ZUM), volume 967 of *LNCS*, pages 510–525, Limerick, Ireland, September 1995. Springer.

[MAS97] L. Mikusiak, M. Adamy, and T. Seidmann. Publishing Formal Specifications in Z notation on the WWW. In M. Bidoit and M. Dauchet, editors, *Proc. Conf. on Theory and Practice of Sw Development (TAPSOFT 97)*, volume 1214 of *LNCS*, pages 871–874, Lille, France, 1997. Springer.

[Mas99a] C. Mascolo. MobiS: a Specification Language for Mobile Systems. In *Proc. 3rd Int. Conf. on Coordination Languages and Models (COORDINATION '99)*, volume 1594 of *LNCS*, pages 37–52. Springer, April 1999.

[Mas99b] C. Mascolo. Specification, analysis, and prototyping of mobile systems. In *Doctoral Workshop in Proc. 21st Int. Conf. on Software Engineering(ICSE-99)*. ACM Press, May 1999.

[MD98] E. Maler and S. DeRose. XML Linking Language (XLink). Technical Report http://www.w3.org/TR/1998/WD-xlink-19980303, World Wide Web Consortium, March 1998.

[MDEK95] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In W. Schafer and P. Botella, editors, *Proc. 5th European Software Engineering Conf. (ESEC 95)*, volume 989 of *LNCS*, pages 137–153, Sitges, Spain, September 1995. Springer.

[MEF00] C. Mascolo, W. Emmerich, and A. Finkelstein. Xmile: An Incremental Code Mobility System based on XML Technologies. In *Poster Session of the. 2nd Int. Symposium on Agent Systems and Applications Mobile Agents*, Zuerich, Switzerland, September 2000.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[MK96] J. Magee and J. Kramer. Dynamic Structure in Software Architectures. In D. Garlan, editor, *Proc. 4th ACM SIGSOFT Conf. on Foundations of Software Engineering*, volume 21:6 of *ACM SIGSOFT Software Engineering Notes*, pages 3–14, 1996.

[MK00] F. Mattern and D. Kotz, editors. *2nd Int. Symp. on Agent Systems and Applications and 4th Int. Workshop on Mobile Agents*, LNCS, Zuerig, Switzerland, September 2000. Springer.

[MPR99] C. Mascolo, G.P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. In O. Nierstrasz and M. Lemoine, editors, *Proc. 7th European Software Eng. Conf. (ESEC/FSE 99)*, volume 1687 of *LNCS*, pages 39–56. Springer, 1999.

[MPR00] C. Mascolo, G.P. Picco, and G.-C. Roman. CODEWEAVE: Exploring Fine-Grianed Code Mobility. Technical Report 00-02, Washington University in St. Louis, January 2000. Submitted for Journal Publication.

[MR98] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.

[MR99] P.J. McCann and G-.C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Trans. on Software Engineering and Methodology*, 8(2), April 1999.

[NFP98] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[Obj97] ObjectSpace. Voyager, 1997.

[OMG95] *The Common Object Request Broker: Architecture and Specification Revision 2.0.* 492 Old Connecticut Path, Framingham, MA 01701, USA, July 1995.

[OMG98a] OMG. *CORBAservices: Common Object Services Specification, Revised Edition.* 492 Old Connecticut Path, Framingham, MA 01701, USA, December 1998.

[OMG98b] OMG. XML Meta Data Interchange (XMI) – Proposal to the OMG OA&DTF RFP 3: Stream-based Model Interchange Format (SMIF). Technical Report AD Document AD/98-10-05, Object Management Group, 492 Old Connecticut Path, Framingham, MA 01701, USA, October 1998.

[OMG99] *XML/Value Request for Proposals.* 492 Old Connecticut Path, Framingham, MA 01701, USA, August 1999.

[OT98] P. Oreizy and R. Taylor. On the role of software architectures in runtime system reconfiguration. *IEE Proceedings – Software*, 145(5):137–145, 1998.

[Pic98] G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, LNCS 1477. Springer, 1998.

[PRM97] G.P. Picco, G.-C. Roman, and P. McCann. Expressing Code Mobility in Mobile UNITY. In *Proc. 6th European Software Eng. Conf. (ESEC/FSE'97)*, volume 1301 of *LNCS*, pages 500–518. Springer, 1997.

[PRS94] R. Pattis, J. Roberts, and M. Stehlik. *Karel the Robot.* Wiley, 1994.

[PS99] L. Petre and K. Sere. Coordination Among Mobile Objects. In P. Ciancarini and A. Wolf, editors, *Proc. 3rd Int. Conf. on Coordination Models and Languages*, volume 1594 of *LNCS*, pages 227–242, Amsterdam, Netherland, April 1999. Springer.

[Res97] S. Ressler. *The Art of Electronic Publishing.* Prentice-Hall, 1997.

[RH98] K. Rothermel and F. Hohl, editors. *2nd Int. Workshop on Mobile Agents*, number 1477 in LNCS, Stuggard, Germany, September 1998. Springer.

[RMI98] Sun Microsystems. *Java Remote Method Invocation Specification*, Revision 1.50, JDK 1.2 edition, October 1998.

[RPZ97] K. Rothermel and R. Popescu-Zeletin, editors. *Mobile Agents: 1st International Workshop MA '97*, volume 1219 of *LNCS*. Springer, Apr. 1997.

[S+95] M. Shaw et al. Abstractions for Software Architecture and Tools to Support Them. *IEEE Trans. on Software Engineering*, 21(4):314–335, April 1995.

[SG96] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline.* Prentice-Hall, 1996.

[SMG94] C. Sperberg-McQueen and R. Goldstein. HTML to the Max: A Manifesto for Adding SGML Intelligence to the Wold-Wide Web. In *Proc. 2nd Int. WWW Conf.: Mosaic and the Web*, page (Electronic proceedings), 1994.

[SMT98] G. DiMarzo Serugendo, M. Muhugusa, and C. Tschudin. An Survey of Theories for Mobile Agents. *World Wide Web*, 1(3):139–153, 1998.

[SN97] W. Scacchi and J. Noll. Process-Driven Intranets - Life Cycle Support for Process reengineering. *IEEE Internet Computing*, 1(5):42–51, Sept/Oct 1997.

[Sof99a] BlueStone Software. XML Contact. http://www.bluestone.com/xml/XML-Contact/, 1999.

[Sof99b] BlueStone Software. XwingML. http://www.bluestone.com/xml/XwingML/, 1999.

[Spi92] J. Spivey. *The Z Notation. A Reference Manual.* Prentice-Hall, 2 edition, 1992.

[SUM96] H. Simpa, T. Uribe, and Z. Manna. Deductive Model Checking. In *8th Int. Conf. on Computer-Aided Verification (CAV'96)*, volume 1102 of *LNCS*. Springer, 1996.

[Sun95] Sun Microsystems. *The Java Language Specification*, Oct. 1995.

[VC99] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In ?, editor, *?* , volume ? of *LNCS*, page ? Springer, ? 1999.

[VCB97] F. Vitali, C. Chiu, and M. Bieber. Extending HTML in a principled way with displets. *Computer Networks and ISDN Systems*, 29(8-13):1115–1128, 1997.

[Vit99] J. Vitek. *The Seal model of mobile computation.* PhD thesis, Computer Science, 1999.

[Web] WebEQ. `http://www.webeq.com/`.

[WF98] M. Wermelinger and J. Fiadeiro. Connectors for Mobile Programs. *IEEE Trans. on Software Engineering*, 24(5):331–341, 1998.

[WF99] M. Wermelinger and J. Fiadeiro. Towards an algebra of architectural connectors. In O. Nierstrasz and M. Lemoine, editors, *Proc. 7th European Software Eng. Conf. (ESEC/FSE 99)*, volume 1687 of *LNCS*, pages 393–409. Springer, 1999.

[Whi96] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.

[WPM99] D. Wong, N. Paciorek, and D. Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, 1999.