A Context-Based Approach for Mobile Application Development

Lukito Edi Nugroho
Ir. (Gadjah Mada), M.Sc. (James Cook)

Thesis submitted in fulfillment of the requirement for the Degree of Doctor of Philosophy

School of Computer Science and Software Engineering
Monash University

March 2001

Declaration

This thesis contains no material that has been accepted for the award of any other degree or diploma in any university or other institution. To the best of my knowledge, this thesis contains no material previously published or written by another person except where due reference is made in the text of the thesis.

Lukito Edi Nugroho

Dedicated to my mother and my late father,
who always gave the best to their children.

Acknowledgements

My first deep gratitude goes to my supervisors, Professor Bala Srinivasan and Professor A.S.M. Sajeev. They have been in great assistance for me when it came to difficult problems. Their encouraging comments often brought my spirit back in enduring hard times.

The Australian Agency for International Development (AusAID), the Engineering Education Development Project (EEDP) of the Ministry of Education, the Republic of Indonesia, and the Distributed Systems and Technology Centre (DSTC), Australia deserve much appreciation for their generous financial and non-material assistance. Without them, I would have not been able to enjoy a pleasant study at Monash University.

I understand that some lovely people have missed me when I was in Melbourne, far away from home. Their great support had a special place in my heart. Mum and sisters, thank you very much.

My life would have not been fun without jokes and cheers from my fellow students. Rosanne, Shonali, Maria, Santosh, Campbell, Le, and Guy have created an entertaining world for me. The parties and the badminton, tennis, and ping-pong games we had have helped me maintain my insanity. Thanks, folks.

My Indonesian friends also deserve many thanks. The Kuncoro and Wibisono families have always welcome me with their warm care. They brought a truly Javanese atmosphere, something that I often missed when I was in Melbourne.

Finally, my deepest appreciation goes to Ermi, Rizka, Ais, and Olly, for giving me their loving support, care, and considerations. Being with them really brings colourful days to me. Once again, being in Australia has not only given me knowledge about computers, but also taught me about family life. It showed me how important they are in my life.

Abstract

Mobile computing is becoming popular because it allows computation elements to be moved or carried around. The breakthrough is partly achieved using mobile applications that allow computation execution to break spatial and temporal constraints.

Mobile application development has to address two key factors. The first is the penetration of physical elements from its execution environment into a program's computation. Handling and manipulation of abstractions of the physical elements in the program is crucial because the penetration is influential to the application's computation. The other factor is the type of mobile entities represented in a program. This is significant as different types of mobile entities require different kind of support.

Many development proposals only offer partial solutions in addressing the two important development factors. They cannot deliver a generic and uniform approach, producing software that cannot handle problems with complex mobility requirements. In addition, the resulting software tends to have complex and non-modular structure, making it difficult to maintain. This problem has to be approached from the conceptual level. This allows an in-depth analysis of the characteristics of the needed support, which in turn provides the basis for a suitable approach for mobility specification and implementation.

A new mobility modeling framework based on the notion of physical environment has been developed. The physical environment is abstracted using the concept of *context*. The framework explicitly separates the *mobil-*

ity and *functionality* aspects by placing them in different abstraction levels. Mobile systems are modeled using a *vehicle* metaphor. The metaphor expresses mobility in terms of *context state* changes within the environment, and uses it as a basis for activating a functionality.

The modeling framework is equipped with a language for specifying the mobility properties and behaviour of an application. The language facilitates *mobility control* to be specified in an abstract, high-level fashion using constructs with context-oriented semantics. The syntax of the language is designed to allow gradual abstraction refinement leading to the implementation stage, where the functionality of the application is realised.

Two implementation prototypes have been built to demonstrate the feasibility of the modeling framework to support the development of mobile applications. The prototypes demonstrate the genericity of models generated from the framework in catering for mobile applications with different mobility support requirements, as well as those operating at different levels. The first prototype is a programming framework for Java-based mobile applications. The framework is developed directly from the specification language by implementing its constructs using Java classes. The second prototype implements support for mobile users on a Linux shell environment. It allows a user to manage task execution based on the location context.

In summary, this research presents a new approach for mobile application development. The approach promotes mobility as an independent abstraction, and highlights the role of modeling and specification of mobility control based on that abstraction. It allows the application of a generic and uniform method to support different types of mobile entities. The method can even be applied using different implementation tools, and in different computing environments.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Mobile Computing and Its Trends

Advances in transportation offer mobility to people. Ease of traveling allows people to perform tasks at different locations and even during a journey. This mobility phenomenon in the physical world has inspired a similar model for computation. *Mobile computing* is a computing paradigm similar to distributed computing in which computation is not restricted to a particular computing location. However, mobile computing goes further than distributed computing. It pushes location-transparent access even further by allowing mobility as part of a computing process.

In distributed computing, the concept of *location-independence* promotes ubiquitous access. In mobile computing, the goal is to explore the dynamics within the ubiquity itself. Portable computers, personal digital assistants (PDAs), and Internet-capable mobile phones are computing devices which allow users to do uninterruptible computing activities while traveling. This puts users in an *always-on* situation where mapping between activities and places is starting to disappear, and human-human and human-computer relationships become continually present [Agr01].

In a different perspective, mobile computing is not only associated with mobile users. Mobility of program code is also exploited. Applets, which are based on the client-server concept, have shown that it is possible and worthy to send program code from one machine to another to do a specific job. In its evolution, some restrictions on applet systems can be removed to allow more flexibility. For example, code can be sent to a remote computer without having to follow a particular communication pattern. Code mobility, in fact, has become an alternative to the client-server model in distributed computing with some performance increases observed in certain areas of applications [Knu95].

Mobile computing is made possible by advances of technology in several areas. In the hardware area, current chip technology makes it possible to manufacture processors that achieve better power-to-performance ratios and are more compact (e.g., the Crusoe family processors [Kla00]). They allow the construction of smaller yet more powerful portable computers and embedded computing devices that can be planted on everyday things such as cars, rooms, air conditioners, microwave ovens, and alarms. Additionally, networking infrastructure is getting ready for reliable wireless communication, facilitating communication from virtually anywhere. Faster network performance and more flexibility are partly due to redesigned network packet transportation mechanisms to suit wireless communication [Per98, CI96, PB96].

Software standardisation has also produced a positive impact on mobile computing. Java [AG97], Common Object Request Broker Architecture (CORBA) [Sie96], Extended Markup Language (XML) [BPSM98] and its descendant Wireless Markup Language (WML) [HTK98] are examples of software technologies that are becoming de-facto standards in their respective fields. These standards offer higher degrees of interoperability to mobile computing applications that work in a heterogenous computing environment, such as the Internet.

Using these technological advances, mobility is no longer regarded as a handicap in doing computing activities. On the other hand, it is now becoming *part* of them. Mobile computing is one of the ways to go to *pervasive computing* where computation occurs in many aspects of a human's physical environment. Automation of daily tasks can be controlled remotely. Personal assistant software can be synchronised with business travel. Traditional human interactions can be replaced by software elements (e.g., in electronic commerce where negotiations and transactions can be done using *mobile agents* [Whi94, GTM+97]). Information need not be carried along since it can be accessed from any mobile position. In short, humans can expect a radical change in interacting with their environment [Mar99].

1.2 Implications of Mobility to Applications

Mobility not only creates a big impact on the way humans do computing, but changes the characteristic of applications that run on a mobile computing environment as well. The most noticeable characteristic is that a movement can be expressed in terms of change of locations. Applications working with mobility are accordingly exposed to such changes. In contrast to a distributed system that hides the location aspect from programmers and users (i.e., location transparency), a mobile computing system must expose the location dynamics as it is often used for reasoning purposes in an application's computation. For example, in a Web-based information system for mobile users, current user location is used to compute the information presented in a browser [VB94]. In such an application, user migration expressed as changes of location becomes part of the application's computation.

In a more general sense, mobile computing applications are required to be aware of other aspects of their environment, not just the location aspect. They are required to possess the ability to detect changes in those

environmental aspects. As mentioned in the previous paragraph, this feature could be used to trigger the execution of some functions to achieve a specific goal. It is most useful in a situation where more physical aspects are becoming the objects of computing. In such a situation, computation can be programmed to penetrate deep into human's environment, retrieving information that can be processed for certain purposes. For example, in an environment populated with intelligent sensors, it is possible to detect the presence of a person, current temperature, network speed connection, and other physical measurements, and then to feed such data to an application which processes it to simulate the physical environment of the user. The information can then be used to decide the application's reaction for a particular environmental situation.

A slight variation to the use of environment awareness is to achieve *adaptivity*, a survival mechanism for an application to keep on functioning in a highly dynamic environment. It is basically a feedback control mechanism. When a change in the environment occurs, it is detected and sent to the application as a feedback which is used to determine the correct operation mode of the application. This mechanism is often employed by applications whose performance greatly depends on the environment. Consider a mobile video conference program. In this application, adaptation techniques can be used to guard the application from network speed fluctuation, low battery power, or transmission noise. For example, if battery power is low, the program can go to a power saving mode, for instance by applying an image compression algorithm to the transmitted image stream.

A mobile computing environment is not only dynamic, but also *open*. The mobile computing paradigm forces machines in a networked environment to accept mobile users or program code. The host machines must allow foreign entities to come and carry out some computation in the location represented by the host within its safety and security constraints. Unlike

a distributed system where the remote communication mode between two sites is completely defined, mobile computing environments can expect interactions with unknown and untrusted entities. In this situation security becomes a vital issue. It is important that such interactions do not harm the mobile entities, the environment, and the underlying system and resources.

Mobile computing is also subject to limited resource availability. Although laptop batteries now have substantially longer life and wireless networks are getting faster and more reliable, they are still far from their desktop equivalent. The main challenge is to allow applications to run in environments with limited resources. Efficiency becomes a key issue, and from the software point of view, this problem is mainly tackled by providing more efficient system support or mechanisms [Mic00, PRWS98].

More compact and smaller portable devices are not always advantageous. From the human-computer interaction point of view, devices with smaller size bring non-ergonomic user interaction problems. Input and output devices must be squeezed to fit into a limited space, preventing users from getting full and flexible control of these devices. Applications running on such devices should consider a proper user interface to allow users to use a new way of interacting with them [SG97, KI98].

1.3 Mobile Applications and Mobility Control

As far as mobility is concerned, there are many types of applications that are related to mobility. These range from traditional, distributed applications that can be used in a mobile computing environment through the help of the underlying network layer (e.g., using Mobile IP [Per98], for instance), to those which exploit mobility through an intermediate layer (e.g., user interface [Ric95]), to those which actually *make use* of mobility in their computation.

It is on the last of those types of applications that this thesis focuses. These are applications whose main computation has a direct relevance to mobility. In subsequent discussions, they are called *mobile applications*. In general, mobile applications can be categorised into two groups: those which *support* mobile users through their ability to operate in a mobile environment, and those which *manage* code migration in their computation. The first category is represented by applications that can react and adapt to changes in their environment, and the second category is exemplified by mobile agent applications.

The concept of mobility introduces a new responsibility to a mobile application. In addition to performing its functional task, the application has to handle mobility-related issues in its computation. This activity is referred to as *mobility control*. Mobility control is needed to specify a migration, or when a mobile application is expected to be aware of and adaptive to its environment. In an awareness scheme, mobility control is needed to tell the application when to execute a functional task (e.g., when the user arrives at a particular location). In an adaptation scheme, it is needed to guard task execution from the dynamics within the environment caused by mobility.

The presence of mobility control in a mobile application reflects a perspective in which the functionality of the application is distinguished from its mobility. Functionality refers to a program's ability to perform a *functional* task. It is associated with *functions*, which are part of the software that provides a core solution for a given problem. In a computing environment where external factors are negligible (e.g., a stand-alone system), the performance of a function is solely determined by its quality, which depends on its internal design (e.g., algorithm efficiency). In a mobile computing environment, however, this statement is not valid. A well-designed function does not always run as expected since it is exposed to a highly dynamic environment that may affect its execution.

The relationship between functionality and mobility in a mobile application is shown in Figure 1.1. A mobile application can acquire inputs related to its functionality or mobility. The former is shown, for example, by normal user interaction, while the latter is captured, for example, using sensor mechanisms. The latter is caught and processed by the control part. Based on the processing result, the control part may tell the functional part to do some action. This could simply mean telling the functional part to do its job, or to make some adjustments before doing the job (illustrated by the vertical upward arrow). However, in some cases the initiative comes from the functional part. It may talk to the controller if during its execution it finds out that some mobility-related issues have to be communicated (shown by the downward arrow). For example, in the information retrieval area, a mobile agent for searching information has the ability to determine the next location to visit if it cannot find the searched information at its current place. This location data is passed to its mobility control component, which is then used to movee the agent to the specified destination. When the agent arrives at the location, the controller notifies the search engine to restart its job.

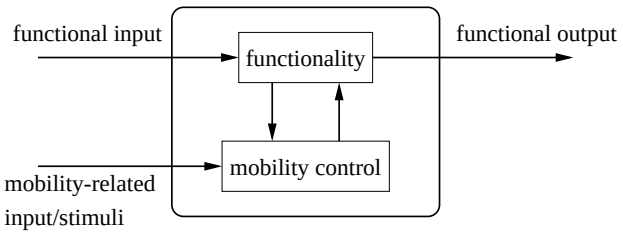

Figure 1.1: Functional and control components of a mobile application

In general, the introduction of mobility control in a computation changes both the structure and the working mechanism of a mobile application. It brings in new issues that have to be addressed in the development of mobile applications. They include representation and manipulation of contexts, facilitating the heterogenity of mobile entities, and realisation of

separation of mobility from functionality. They are discussed in the following section.

1.4 Issues in the Development of Mobile Applications

Many mobile applications capture the notion of location in their programs. The location notion gives a mobile application the sense of *being somewhere*, and when combined with the application's functionality, it describes the spatial dimension of the problem-solving activity performed by the program. In fact, location is just one example of environmental elements whose abstractions penetrate into an application's computation to form conceptual domains that orthogonally surround the computation. Such an abstraction is called *context*[1], and has an important role to describe the relevance of an environmental element to the mobility of entities represented in the application.

The relevance of a context to a mobile application is shown by its ability to indicate the mobility of an application. Using the concept of context, mobility is not only expressable in terms of location (i.e., spatially), but in terms of other environment elements as well. To give examples, mobility could be expressed in terms of network connection speed (i.e., a migration from a wired network node to a wireless network node would cause speed change), resource availability (i.e., different computers have different CPU power, memory, disk, and other computing resources), or surrounding people (i.e., movement changes the neighbourhood, represented by the people logging on the same machine). Of course such changes can be triggered without migrations actually taking place, but it does not really matter since a

[1]The term *context* here refers to real-world, physical elements such as locations or network speed, rather than internal program execution environment (e.g., memory heap, execution stack, or program counter).

mobile application is only interested in the changes, whether or not they actually involve migrations as well. Figure 1.2 shows the mental image introduced by the presence of a context to computation. It defines possible "topics" mentioned in a mobile program, expressed using terminologies describing physical elements referred by the context.

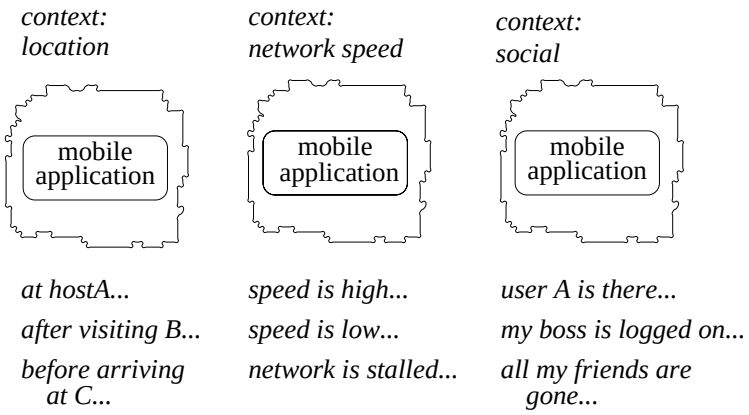

Figure 1.2: Contexts surrounding a mobile application

The association between a context and a functional task has to be reflected in the development of the application. Programmers have to put functional computation in the context's perspective. Therefore the focus of programming of a mobile application is on organising contexts used in the application and associating them with appropriate functional computation. This is basically different from programming of other types of applications, which focuses only on the specification of functional computation.

The second issue in mobile application development is raised by the presence of multiple types of mobile entities represented in mobile applications. With mobility is fully exploited, entities representing people and code can become mobile, and both types of entities can appear in the same application. Traditionally, development tools are designed to support only one dominant type of mobile entities. For example, tools designed for mobile

agent applications concentrate on support for code mobility. The tools focus on how to make code representing an agent movable from one place to another. The support is usually realised using specific commands implemented as function or method calls [Obj99, LO98] and special execution environment installed on machines in a network. On the contrary, these things are mostly irrelevant to a network-aware application that supports mobile users. What is more important is probably a clean implementation of an asynchronous mechanism for monitoring and reporting network speed fluctuations. When both mobile agents and mobile users are present in the same application, a particular tool may not be able to handle all programming needs that arise from this situation.

The above illustration shows the narrow scope of mobility support provided by development tools for mobile applications. Such a narrow scope stems from the failure to view mobility as a unifying concept for performing tasks in a problem-solving framework. Mobility is associated with a *principal* or *actor* (who/what experiences a migration) representing a mobile entity, and support is provided on the basis of the mobility support requirements of the actor. In situations where mobile entities with different requirements are present, a partial solution is no longer suitable, and the need for a more general approach becomes apparent.

The third issue comes from a software engineering perspective, and is related to the architectural support provided by a programming language to develop a mobile application. A program is a construction of software components written in a particular programming language. The structure of a program is dictated by the architectural framework imposed by the language. An architectural framework defines how components can be composed to build a complete program. Like building a physical construction, given a set of linguistic constructs, it specifies what can and what cannot be done with the constructs during an implementation process.

The architectural framework set by a generic programming language may not be able to represent new abstractions introduced by environmental elements. When context abstractions need to be expressed using existing linguistic constructs, programmers have to manually work out the representation. This often forces programmers to disregard the clean architecture that comes with the language, creating a program that has non-natural component composition. The resulting program is usually hard to understand and maintain. A classic example of this phenomenon is the *inheritance anomaly* problem posed by concurrent object-oriented languages [MY93]. The problem states that inheriting a concurrent class often requires modifications of the parent class, which violates the principle of inheritance itself. The ability to recognise and accommodate contexts handling and manipulation is therefore crucial in mobile application development.

Finally, programming of an application's functions is often done independently of specification of mobility control. In many cases, functional components could exist even before the program that makes use of them is written. For example, in the mobile trader example given in Section 1.3, program components representing the trading functionality may have been developed long before the trader agent program is written. Perhaps the trader agent program is an improved version of another program with the same functionality that uses a client-server approach. In such a situation, the designer of the original functional components may not be aware of any possibility to make the components mobile. This shows the need to decouple the development of functionality from that of mobility. It should be easy to turn a function, data, class, or any other linguistic constructs to their mobile version without messing up with their original semantics.

Orthogonality between functionality and mobility is also applicable in the other direction. Mobility control should be independent of the functionalities it operates on. For example, specification of a travel itinerary should

be independent of that of the action executed in locations visited during the travel.

1.5 Motivating Factors and Area of Research

As mobile computing is becoming a common style in using computers, mobile application development is likely to get its momentum as well. From the software engineering point of view, mobile application development can be considered as a development activity that specifically works on mobile applications. A properly supported development process is required to tackle the issues mentioned in Section 1.4. Identification of support requirements, as well as the provision of the support, are the main motivations of this research.

Mobile application development is a research area that has not yet received much attention. Compared with other areas, such as hardware design for mobile computing or provision of necessary network infrastructure, research on this area is one step behind. Mobile applications have not been able to exploit the full potential of mobility, since specific requirements which allow such an extensive exploitation have not been well understood yet.

With the era of pervasive computing coming fast, the need for high-quality mobile applications also rises accordingly. More mobile applications are required to be *allround*, in the sense that they have to handle different kinds of mobile entities with different mobility support requirements. Moreover, mobile computing is moving away from using network-based locations as the only way of expressing mobility. Mobile applications are starting to make use of other aspects as well, such as network speed, office room, and social environment.

Addressing the above requirements using the traditional development model would not yield efficient and effective solutions, due to the partial nature of the approach. The approach does not consider mobility as an in-

dependent abstraction. Instead, it views mobility as a *dependent attribute* of a mobile entity, and support is given on the basis of the programming requirements pertaining to the entity. Moreover, the traditional approach departs from a development model for non-mobile applications which does not recognise the situation where programmers have to deal with abstractions of physical environment elements. In all cases where no direct implementation support is available from the programming tool (e.g., the tool is not suitable to implement the mobility requirements of the mobile application), non-natural programming workaround is needed. This situation usually leads to inefficient, complex, and hard-to-maintain program code.

In a broader perspective, difference in acceleration of research progress between the software and hardware areas introduces a situation similar to the early history of software engineering (or programming in a narrower scope), where an overall progress acceleration in mobile computing area cannot be maximised due to the slower progress of the software side. The research described in this thesis is an effort to bridge the gap. It endeavours a number of areas in search for a better understanding of mobile systems, particularly their characteristic and the support required to handle mobility. This knowledge can then be used to design a suitable approach for developing mobile applications.

The problem of partially supported mobile applications has to be approached from the conceptual level. This is because core and essential aspects of systems manipulated by mobile applications have to be identified so that proper support can be provided. Modeling of mobile systems is a prospective starting point for this study, because a model can provide a view of such a system from the required level.

Modeling aims to describe the characterising properties of a mobile system. It lists mobile entities that form the system, defines functionalities and when/how they are activated, and states the relationship between mobile

entities and their environment. In short, they form components required to control mobility in the system. Since these aspects are exactly what mobile applications deal with, it is therefore suitable to promote modeling as the initial stage of an application development effort. Modeling is essential in establishing requirements specifically related to mobility, which have to be realised in the implementation stage. It becomes the focus of the research since a good quality mobile application software can only be created if it is developed based on a strong ground that sufficiently addresses mobility-related issues.

A model of a mobile system can be represented in many ways. A model representation can then be used as a basis for application design and implementation in further development stages. It is possible to directly implement a model representation using a programming language. However, such a straightforward approach has a drawback. A model description written at the model level is very abstract, while programming at the implementation level deals with detailed abstraction. Such a gap can obscure the novel aspects of the model during abstraction refinement process. A direct implementation effort would easily miss these aspects and prevent an implementation program to take advantages from them.

To bridge the gap so that implementation effort can be carried out smoothly from a given model, a specification tool with sufficient expressive power needs to be developed. A tool in the form of a language fits this requirement. Such a specification language has an important intermediary role in the development of a mobile application. On one side, it is used to represent a model. In performing this role, the language has to be generic and abstract. Its abstraction level should allow developers to specify the essential aspects of a mobile system without having to be restricted to specific implementation issues (e.g., programming paradigm, language, and style). On the other side, the language should provide helpful guidelines for an

implementation effort. In particular, it should narrow down the abstraction gap to facilitate a gradual abstraction refinement process in the development stages beyond modeling.

Using the specification language, modeling is actually an activity of defining mobile entities and writing specification of mobility control for the entities. The novel aspect here is that mobility control is specified at an abstract level. Moving mobility control specification to a stage as early as modeling promotes genericity that allows the model to be used in a wide range of application areas.

As a demonstration of usability, the research also explores the implementation side. The purpose of the work is to assess the applicability of the model-based approach to support mobile application development in different computing environments. From the modeling level, two different directions are pursued. The first is to follow the programming path where mobile applications can be developed from a specification using Java programming language. The other direction of implementation operates at the operating system level. The model-based approach is used to assist the development of a mobile application running on the Linux shell environment.

1.6 Contributions of the Thesis

The thesis contributes several novel concepts in mobile application development. Two implementation prototypes have also been developed as research outcome. The contributions of the thesis are shown as follows.

1. The thesis proposes a modeling framework for describing system properties on which mobile application development can be based. The framework is based on a simple transportation metaphor and recognises physical environment as an influential factor in a mobile application's

computation. To the best of my knowledge, it is the first modeling framework that incorporates the notion of physical environment for the purpose of mobile application development.

Genericity is another highlighted aspect of the framework. The framework offers a uniform view which can be applied on the development of mobile applications with different mobility support requirements. Genericity also leads to a flexible implementation of mobile applications. Such an implementation can be tailored to suit specific circumstances, such as the implementation environment, the development paradigm, and the availability of programming tools.

2. The research also results in a specification language. The language is used as a specification tool to describe a mobile system. Using the language, specification of mobility control can be conducted as early as the modeling phase. At the same time, the language provides guidance for the implementation of the system, allowing a model's abstractions to be refined gradually in the development stages beyond modeling.

3. The spin-offs of the implementation work contribute new ideas in supporting mobile computing in specific environments. The Java-based implementation breaks the tradition of the mainstream style of Java-based mobile application development which concentrates on code mobility by introducing the capability to cater for user-oriented mobility. The lessons learned from the implementation also suggest required linguistic features to naturally support implementation of mobility control. Implementation in the Linux environment extends the operating system's features with user-oriented mobile computing capability.

In summary, the research proposes a new approach for mobile application development based on modeling and specification of mobility control. The novel feature of the proposed approach is that a development is based on

a well-defined model with generic properties that can be applied to any type of mobile applications. The modeling framework can be used to complement the conventional software development paradigm. The framework enriches current development practices with specific features that address the unique characteristic of mobile applications.

1.7 The Structure of the Thesis

There are seven chapters in this thesis. The contents and the organisation of the chapters are shown as follows.

Chapter 1 provides background and introduction material about mobile computing. It describes general issues in mobile computing and their impacts on mobile applications. It also explains the motivations behind the research and justifies their relevance.

Chapter 2 contains a literature survey on issues that are relevant to mobile application development. The survey starts with a discussion on mobile computing as a computing paradigm. The main part of the survey provides some analysis on current approaches for mobile application development, particularly on how user and code mobility are supported. At the end of this chapter, the research goals are revisited to justify the position of the research among other relevant work that has been done or is currently active.

Chapter 3 contains the first part of the thesis' main contribution. The chapter starts with a discussion on some key issues found in mobile applications. Then it describes the proposed mobility model, followed by the presentation of the formalism for the model. A discussion on the highlights of the model concludes this chapter.

Chapter 4 explains the specification language. It begins with a description about the rationale behind the design of the specification language. Following this, the syntax and semantics of the language are presented, and specification examples are provided.

Chapter 5 describes an implementation that maps specification language constructs into Java constructs. This chapter also discusses some lessons learned from the implementation.

Chapter 6 describes a different implementation course. The case is the implementation of the Mocha model on a Linux environment. A tool prototype is developed to provide location-awareness that allows mobile users to manage command execution based on their locations.

Chapter 7 presents the conclusions of the research, and suggests some possible research work that can be done in the future to improve or extend the outcome of the current research.

Chapter 2

Mobile Applications and Their Development

2.1 Introduction

The topic of this thesis covers two broad areas: modeling and development of mobile applications. The modeling part focuses on devising a modeling framework for mobile applications development. To design a representative model, it is important to have a good conceptual understanding of different aspects of mobile computing so that its essential properties can be identified and represented.

The modeling framework should also allow a model to be placed into the perspective of software development. A model is viewed as a specification of mobility-related requirements, upon which a software development process is based. Software engineering considerations should be applicable so that development stages beyond modeling can benefit from it.

This chapter presents a survey to provide a background for the areas of the research reported in this thesis. Due to the broad coverage of the topic of the research, the survey takes a top-down approach. It starts with Section 2.2 that presents a discussion on mobile computing as a computing paradigm and a comparison with other related computing paradigms. This is followed by

Section 2.3 that explains general support for mobile computing. A general perspective of mobile computing from the distributed computing point of view is provided. The main issue here is how mobility can be exploited and manipulated so that computing activities can be conducted on top of a distributed computing infrastructure. The survey presented in these two sections portrays mobile computing from a broad angle. Its objective is to identify essential properties that characterise mobile computing systems.

Section 2.4 looks at applications running on mobile computing environments. The focus of the survey remains the same as that of the previous section, that is, how to turn mobility into a supportive computing factor. This time, however, an application's point of view is used. Different approaches for handling mobility-related issues are presented. The survey gives an idea on implementing different types of mobility support *within* an application.

Finally the survey looks at the development aspect of mobile applications from a software engineering perspective. Development issues raised by the presence of mobility-related concerns in mobile applications are discussed. The objective of the survey is to raise awareness of these issues so that they can be addressed properly.

A set of statements on the research problems conclude this Chapter. These statements justify the relevance of the research presented in this thesis.

2.2 Mobile Computing and Related Concepts

The term *mobile computing* refers to a computing activity that is in some ways related to mobility. In many situations mobility is associated with a user, and mobile computing therefore reflects the ability of a user to do computing activities during his or her travel. Some researchers use the term *mobile computation* to denote computation elements that can migrate to

different machines [Car99]. However, from this thesis' point of view, the distinction between the two terms is not significant, since the emphasis is on the mobility aspect rather than on the principal of migration. In subsequent discussions, the term *mobile computing* can be used to denote either situation.

The definition of mobile computing implicitly states that it has a close similarity to distributed computing. Both of them are based on the concept of location distribution as a determinant computing factor. Computing elements are bound to a spatial environment formed by a number of networked machines located in geographically different areas.

The aim of distributed computing is to mask location heterogenity and distribution. A distributed computing system tries to hide the location of program components from the user to achieve the notion of *access from anywhere*, that is, from any network node in the distribution scope. However, the concept can only be applied with coarse granularity, because the computers that form the access points are normally distributed sparsely in geographically different locations. Mobile computing refines the granularity of distributed computing by increasing the density of access points in a distributed computing scope. Using a distributed application during a journey between two locations can be viewed as distributed computing over an infinite set of locations between the two places.

Mobile computing is, however, different from distributed computing in other aspects. Distributed computing has static characteristic. Once an application's elements are set in their location, they are fixed there. On the other hand, mobile computing elements can freely roam a network.

Another difference between the two computing paradigms is related to their environment. The performance of a distributed program is affected by certain environmental elements (e.g., network connection), but the significance of the influence is not as much as in a mobile application. Migrations

change the location of an entity, and this can introduce some dynamics to the traveling entity. Different environment settings, different security measures, and meeting with other mobile entities are examples of such dynamics caused by migrations. In a mobile computing environment, it is often necessary to make an application aware of recurring changes. This is, in fact, the biggest difference between mobile computing and distributed computing. In mobile computing, information related to the dynamics of the environment is often required by an application to reason its computation.

The idea of seamless computing regardless of the user's physical position is not very new. The term *ubiquitous computing* was first coined by Mark Weiser to describe a situation where a user is constantly interacting with a large number of computing devices [Wei93]. Weiser stresses that even though the interaction is intensive, it should not make the computers the focus of attention. In other words, they should disappear from a user's awareness when he or she works with them so that an activity can be conducted without the need for high technological skills. He argues that this situation can be achieved only by applying radical changes in the relationship between humans and computers in which computers take the form of artifacts which mimics common things in human's daily life (e.g., papers and pens) and work closely like them. By disguising computers in the forms human are familiar with, they become mentally transparent to people who use them.

After Weiser's seminal paper [Wei93], the effort to realise seamless computing started to grow. For example, Abowd [Abo96] also looks at the same issue, but his focus is on applications running on an ubiquitous computing environment. Such an application has to be able to take over human's function in capturing events that surround him or her, memorising the important ones, and recalling them in the future. Furthermore, the application should also have the ability to integrate information brought along with the events that often come from different sources.

A different view of ubiquitous computing is given by Dearle [Dea98]. He proposes the concept of an ubiquitous environment that allows mobile users to perform computation on top of it. An ubiquitous environment consists of three layers: a *view* which defines what a mobile user sees during his or her journeys, a *platform* which consists of hardware and software elements that implement a view, and some *servers* that are designed for general-purpose repository. In this architecture, user migration can be expressed in terms of view migration (i.e., the same view can be reproduced by a different platform in a different location) or platform migration (i.e., the platform also migrates, possibly involving a network connection change).

The difference between mobile computing and ubiquitous computing lies in the way they perceive seamlessness of computing activities. Ubiquitous computing tends to focus on the result or effect of the *interaction* between a user and an underlying computing system. Ubiquitous systems can be built by developing appropriate hardware, network, and software infrastructure. Mobile computing, on the other hand, focuses the *effects of mobility* to applications. Development of mobile computing systems normally concentrates on the provision of appropriate support to handle such effects.

Weiser's vision of ubiquitous computing [Wei93] converges with current situation where more portable and small-scale computing devices are embedded in things commonly found in human life such as cars, clothing, and household devices. With these devices populating our physical environment, computing is naturally everywhere. It is common to find computation penetrating many aspects of human life; in other words, it becomes *pervasive*. When embedded processors turn out to be the majority of all computing devices produced [Ten00], pervasive computing changes the nature of human-computer interaction. For example, such interaction becomes implicit, exhibited by various kinds of automation performed by embedded devices [Mar99].

A pervasive computing environment not only facilitates user-oriented mobility, but allows exploitation of code mobility as well. The latter form of mobile computing has a potential application in controling computing devices remotely. Remote controling via mobile code offers more flexibility because of the programmable nature of the code. For example, the user can adjust the lighting of his or her house to adapt to outdoor light intensity, or even to follow a certain pattern to trick thieves.

Code mobility is not a new idea. It has been used for different areas of applications, such as to submit batch jobs remotely [Bog73] and to control printers using the PostScript language [Sys85]. This topic enjoys a more structured approach in the distributed operating system area. In this area, researchers have been trying to support the migration of active processes and objects at the operating system level [Nut94]. Process migration is mainly used as a load balancing technique in distributed systems [Fre91, BLL92, BGW93]. It allows load balancing to be performed dynamically by migrating process code and other objects that represent its execution state to different hosts.

Code mobility gains its momentum through the *applet* concept introduced by the Java programming language [AG97]. In an Internet-based computing environment, an applet is a piece of code that resides in a Web server. When a Web browser downloads a document that contains the applet code from that server, the code moves along with the document. If the browser supports the execution environment required by the applet, it will be executed in the browser. The effect is like having a program running in the browser. Through applets, the concept of mobile code has become popular because migration is performed at the user level. User-level migration allows a user to have control on code migration. Coupled with the cross-platform characteristic of Java, this feature offers a simple way to leverage traditional applications to work in a distributed computing environment.

The mobility of an applet's code is, however, limited. Code migration is performed one-way (i.e., from a Web server to a Web browser), because the applet system follows the client-server paradigm. To fully tap the power of code mobility, code migration should not be constrained to any communication pattern between a source and a destination. Instead, code should be freely migratable to any computer with appropriate support. This mobility model is exploited by *mobile agent* applications. There is still no authoritative definition and uniform understanding among researchers of the concept of *agent* itself [FG96]. However, as far as this thesis is concerned, an agent is simply a piece of code that is programmed to do a specific task with a degree of autonomy. A mobile agent is therefore an agent that has the ability to move and has some autonomy in making decisions[1].

The concept of mobile agent improves the flexibility of the client-server paradigm exhibited by the applet concept. A mobile agent can be transported to a target machine without being constrained to the functionality of the machine (i.e., irrespective of whether it acts as a client or server) . This is because a mobile agent is self-sufficient, it has the required know-how for its execution. All it needs from a destination host is the resources [FPV98]. This makes mobile agents an attractive alternative for pervasive computing, since a user can send his or her agents to run on embedded computing devices without having to set up certain services in the devices prior to the execution.

The autonomy of mobile agents can also be exploited. A mobile agent can be programmed with knowledge to respond to certain situations that occur during its journey. This feature is useful to create agents that act *on behalf* of their users or owners. Potential areas of applications of this concept include electronic commerce. In this area, mobile agents can be used to simulate traditional trading systems. Interactions between traders and

[1]The term *autonomy* should be interpreted from an agent's internal view. In this perspective, it refers to a reasoning mechanism built into an agent's computation.

buyers open up various scenarios that normally require autonomous actions from both sides. The difference is that communications, negotiations, and transactions are now performed by mobile agents [DNMMS99, GTM+97].

To reiterate the material presented in this section, the concept of mobile computing has been presented along with the related computing concepts. The key characteristic of mobile computing is the presence of mobility abstractions in an application's computation, *close enough* to introduce some *effects* on the computation. It is not important whether the movement of a mobile entity happens externally to the computation (as in the case of a mobile user with his or her application), or internally as the computation itself moves (as in systems with code mobility).

The effects brought by mobility to applications vary. Mobility can bring negative impact as it can reduce the performance of an application. On the other hand it can also bring positive impact, since the behaviour of an application can be adapted to the current situation of the environment. Therefore it is necessary to properly address the mobility of an application. The next section looks at this issue from the distributed computing point of view.

2.3 Working with Mobility

This section discusses the implications of mobility on a computing system and how they can be supported, using the distributed computing point of view. Although the main focus of the discussion is the effects of mobility at the application level, some of those which affect the underlying network level will also be explained to give the idea of the overall implication of working with mobility.

2.3.1 Mobile Networking

In mobile networking, the physical location of a mobile computer no longer determines its network address. From the network point of view, there are three challenges that have to be addressed [IK96], and they are:

- How to make a network know the current location of a mobile host,
- How network messages can be routed to a mobile host, and
- How to improve the network transport performance over a mix of wired and wireless networks.

The Internet community addresses the first two problems by aiming a new standard called *Mobile IP* [Per98, Per97] which extends the Internet Protocol (IP) mechanism to work with mobile hosts. In systems based on the Transmission Control Protocol/Internet Protocol (TCP/IP), transport-layer connections are determined by a quadruplet number that specifies the IP address and port number of both connection end points. The IP numbers have to be maintained, otherwise the connection will be lost. When a mobile host migrates and reconnects at a new network, its IP address changes. To maintain connectivity, Mobile IP allows a mobile host to use two IP addresses. The *home address* is a static IP address and is used to identify TCP connections. The *care-of address* changes at each point of attachment and is regarded as the effective address. This address must be registered with a special node called *home agent* every time the mobile host moves. In principle, a connection can be maintained by redirecting network packets to the care-of address by the help of the home agent. The mechanism is called *IP tunnelling*, and is implemented by constructing a new IP header which contains the care-of address, encapsulating the original home address.

Equipped with the ability to do wireless communication, mobile users can set up a temporary wireless network among their mobile computers that

does not require existing network infrastructure. Such a network is called an *ad-hoc* network. An important characteristic of an ad-hoc network is that its topology changes over time and there is no notion of administrative nodes. This poses a routing problem because existing routing protocols [Hen88, SS80] depend on that function. A possible technique to solve the problem is to operate individual mobile hosts as routers [PB96]. The main idea is to add a self-starting behaviour to the routing mechanism. When a mobile host moves, it announces its new network topology, and triggers an update of routing table in other hosts reflected in the new path.

One main performance problem in TCP/IP-based systems in mobile networking is the incompatibility between the TCP mechanism [ISI81] that is responsible for transporting network packets and the wireless networks where they operate on. The TCP mechanism automatically slows down when it encounters increases in delays, and interprets them as triggered by network congestions. By reducing the transfer rate, it expects that the congestion will be discharged. In wireless networks, delays are not always associated with network congestion. They may be caused by normal actions of mobile users, for example, when a mobile host switches from one network cell to another. Such a false interpretation can cause the TCP mechanism to cause unnecessary slow downs that reduce the performance [CI96]. To have a mechanism that works both for wired and wireless networks, one may have to create a new protocol, which could be impractical. One solution is to use an indirect model similar to that used in Mobile IP, by splitting the TCP protocol into two parts that address the fixed and mobile connections of a mobile host [BB96]. In this approach, the protocol that serves the fixed part does not have to be modified and a new protocol can be specially designed to serve the mobile link.

2.3.2 Applications and Mobility

Given a proper network-level support, it is conceptually possible to run a normal distributed application in a mobile environment. For example, using a Mobile IP implementation (e.g., Solaris Mobile IP for Solaris machines [Gup98]), one can run programs such as telnet, ftp, and many others as if they run on static network connection. However, mobility does not only affect computing systems at the network level. At the higher level, it also introduces some consequences to some aspects of applications running on a mobile environment, as shown in Figure 2.1.

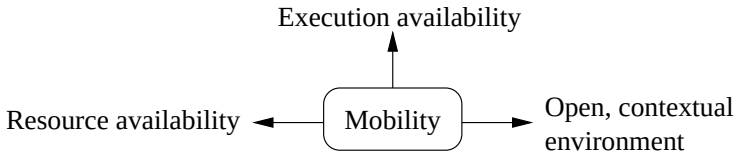

Figure 2.1: Aspects of applications affected by mobility

Execution Availability

Any computer program needs a proper execution support system, and applications running on a distributed environment are no exception. In its simplest form, an execution support system for these applications consists of two parts: *execution engine* and *distribution layer*. An execution engine is a software layer that abstracts an execution environment. It encapsulates low-level execution mechanism such as task scheduling, CPU allocation, and resource management. In Java-based systems, for example, it is implemented by a virtual machine that sits on top of the operating system [AG97]. A distribution layer controls connectivity between application components. It is normally implemented as a suite of network protocols and communication techniques. Availability of both parts of an execution support system to an application guarantees its *execution availability*.

In mobile computing, the absence of an execution support system can bring *blackout periods* to applications. A blackout period is a time period when an application cannot be executed due to the unavailability of an execution support system. Such a situation is typically encountered by a mobile user who runs an application on a traditional distributed computing infrastructure (e.g., using static networks). Although the user may be able to keep working while being mobile through wireless connection, he or she is forced to terminate the execution when he or she is out of the area of coverage. In this situation, the user has to find a new node and reestablish the network connection before resuming the execution. This temporary blackout period is due to the unavailability of the distribution layer when the mobile user moves away from the current node. Techniques discussed in Section 2.3.1 can help to maintain the distribution layer, allowing the user to have continuous execution availability during his or her migration.

Another way to preserve a distribution layer is through a technique called *teleporting* [Ric95, WRB+97]. In teleporting, a user can tell the display associated with a program to move and follow him or her, and rebuild itself in the new location. Teleporting is designed to work with applications running under an X Window system. It inserts an additional X proxy between a client application and an X server. The proxy performs a display indirection whenever the user moves. When this happens, any communication between the client application and the original X server is caught by the proxy and redirected to another X server in the new location. Using teleporting, patching the distribution layer availability problem is done at a higher level (i.e., the user-interface level).

In general, provision of seamless execution availability for applications with user-oriented mobility mainly concentrates on the distributed layer aspect. It is mostly unnecessary to work on the execution engine, because it is always available at any time (i.e., it migrates along with the applications

in the user's portable computers). On the contrary, applications with code mobility need to deal with execution engine availability. When code moves to another location, it leaves its current execution engine. A similar execution engine has to be available in the remote location to accept the incoming code.

For applications with code mobility, the idea is to extend the distributed computing infrastructure to allow *code*, instead of human, to log on a machine and do some work. This is not a trivial problem because traditionally execution engines running on the machines are tightly coupled to the machine's underlying platform (e.g., hardware and operating system). In this environment, execution management assumes that code, data, and resources are all localised in the same machine. In the implementation of a mobile code system, programs' address spaces can span over machines connected in a network, so a proper referencing mechanism is needed.

When program code is migrated, local bindings among objects are lost. In this situation, migration breaks local references to objects, making them inaccessible by other objects. The problem can be managed using different ways. *Network references* have the same function as local references, only they work across network connections. Network references can be implemented on top of a proxy-based mechanism, such as forwarding pointers [IDJ91]. A simpler approach is to use object replication (i.e., to create a copy of the object in the new location). However, replication is subject to integrity constraints, which can be very expensive in highly mobile systems.

Because code migration is performed in the context of an application execution, it must be possible to resume the execution in the destination node. In this situation, how execution state is handled becomes an important issue. There are two approaches concerning this issue [FPV98]. In the first strategy, execution state is migrated along with the code. This approach is called *strong migration*. It allows execution to be continued in

the new location from the point where it is suspended before the migration. This situation is similar to process migration at the operating system level, only strong migration is performed at the application level. The other approach, called *weak migration*, is simpler to implement. Only code is sent, but not execution state. At the destination, a migrating thread of execution is reactivated by restarting it.

Open and Contextual Environment

In the traditional sense, the term *environment* is normally associated with the execution of an application, and refers to the internal execution mechanism. In mobile computing, a new notion of environment is introduced. It refers to the *physical* surrounding of an application. It represents the actual environment of a user, or the external environment of a migrating piece of code. Making physical environment known to an application is significantly important, because it provides the application with a direct interaction with the real-world where the application is running.

Section 1.4 introduced the concept of *context* to represent the abstraction of a physical environment element in a computation. The context abstraction can represent quantitative elements (e.g., IP address, network connection speed, or people in the neighbourhood), or qualitative elements (e.g., are these people colleagues or bosses). A context can be built from an individual abstraction, but in a complex set up, one can have a context made up of several individual abstractions of physical elements. For example, a *workspace* context can be defined by the combination of room number, people sitting in the room, and the purpose of the room (e.g., working room or class room). Information about these elements is relatively easy to capture, for example, from system configuration and log files. The challenge, on the other hand, is to provide a seamless integration of people, computation, and physical reality, for instance, by using the concept of *smart space* [Mar99].

A migration changes an application's context dynamically. *Context-aware computing* tries to exploit context dynamics in a computation. Research in this area has been concentrated on the effort to capture the notion of physical location and represent it as a computation abstraction, which can be used as the basis for programming of specific actions. Location-aware computing has found a wide range of application areas, including Web-based information systems [VB94], virtual guide systems [AAH+96], office environment [WJH97], and computer-supported collaborative work (CSCW) [BSHB98]. Special devices with sensory mechanisms to detect current user location have also been designed to support the operation of location-aware applications [SAG+93, WHFG92].

Mobile computing also requires its environment to be *open*. Migrations implicitly require that locations can accept entities that do not originally belong to them. An acceptance has to be based on a prudential principle so that it does not bring hazards to the incoming entity, the underlying system in the destination location, and all objects that may be affected by the visit.

Vitek, et.al. [VST97] identifies five different areas in a mobile computing system where security measure should be imposed: 1) network transfer, 2) authentication and authorisation, 3) location host and operating system, 4) execution engine, and 5) mobile entities. Data transfer in the network layer must be secured to avoid eavesdropping. Secure data transmission is normally approached by cryptographic techniques, for example, by using Secure Socket Layer (SSL) [FKK96]. Authentication and authorisation are required to ensure that the incoming party is the actual one, and if so, it is granted certain access rights. Kerberos [SNS88, NT94] is a widely-used authentication system that can be used for this purpose. Security issues on data transmission and access are typically found in distributed computing, and user-oriented mobile computing also shares the same problems.

The other security issues are found in applications with code mobility. An operating system and the resources it controls has to be protected from unauthorised access by foreign code running on an execution engine. This is normally approached using an access control mechanism employed by the execution engine (e.g., the domain-based protection mechanism in Java [Gon98]). The security issue regarding an execution engine is two-fold [VST97]. The most important thing is that access by foreign code must adhere to a well-defined interface to protect the execution engine. On the other hand, the execution engine has to be trusted so that it does not do something harmful to the code running on top of it. Finally, interaction between code elements must also be secured. It it necessary to prevent malicious access at the application level.

The openness of a mobile computing environment poses an interoperability problem as heterogenous parties can interact with each other. Heterogenity issues regarding platforms and programming languages have already been taken care of by introducing new standards for a new distribution layer architecture that masks the heterogenity [Sie96, Red97]. However, these standards do not handle diversity in high-level communication. In real-world situations, communications between mobile entities require more than just a common language. Both parties have to share common knowledge about the topics being communicated. Research on the semantics aspect of interoperability has approached this problem using knowledge sharing between communicating parties, which includes the construction of a *lingua franca* for applications that speak in different languages [FLM97, GF92]. Such a very-high-level language has the purpose of achieving high-level interoperability, much the same as a distribution layer in the lower level.

Resource Availability

One of the basic challenges in mobile computing is dealing with the availability of computing resources [FZ94]. In portable devices and wireless networks, CPU power, battery energy, and network connection are luxury resources in mobile computing, compared to their desktop equivalent. Scarcity of resources occurs at the hardware level, and this problem is addressed by designing more efficient hardware such as longer battery life or energy-saving processors (e.g., Crusoe family processors [Kla00]).

At the application level, the same spirit in improving the efficiency is exhibited by various attempts to create software systems that consume fewer resources. For example, a lean execution engine for Java has been designed for consumer and embedded electronics [Mic00]. It is an implementation of Java virtual machine that can run on devices with less than one megabyte of memory. In the network communication area, software agents are used to improve transfer efficiency in wireless networking systems. A user agent is placed in the network, and the purpose of the agent is to shift some processing load from a mobile device to the network itself [PRWS98]. The result is streamlined network streams that yield better bandwidth utilisation.

Lack of computing resources may cause a performance penalty or even failure in application execution. This situation is indeed unacceptable, but because it is not always under the user's control, it is sometimes unavoidable. Applications need to be *adaptive* so that appropriate actions can be taken when unexpected situations occur. An adaptation mechanism can be put on the application's side, the system's side, or as a collaboration between the two [Sat96]. Imposing adaptation mechanism on the application side means the application takes full control in determining its adaptive behaviour. This feature is exhibited, for example, by network-aware applications that can monitor and adapt to changes of network speed [ARS97]. This is done by

querying a supplied distributed network monitor and using the result to adapt its behaviour. At the other extreme, the adaptation mechanism can be made transparent to applications by shifting it to the system's side. This means existing applications can be executed without any modification. Examples of such a system are Coda [KS92] and Ficus [HPRP92] filesystems, which use caching techniques to allow mobile users to keep working despite disconnected operations (i.e., no network connection).

An effort to support mobile computing has also been pursued at the operating system level. The work by Bender, et. al. targets the resource availability problems [BDD+93]. The approach taken is to modify the kernel of UNIX operating system to include a power management device driver, which allows better power utilisation due to the highly dynamic power usage patterns. This is done through a mechanism that allows a user to save the system state at any point, to have smoother usage patterns.

In a different perspective, compact size requirement of mobile devices can also be seen as a restriction that affects user interaction. Normal user interaction techniques cannot be used due to ergonomic reasons. Some new interaction techniques approach the problem by exploiting human nature in working with small things. For example, artificial pointing devices such as mice are replaced by fingers [KI98, SG97].

The popularity of Internet as a giant information repository has also attracted efforts to allow mobile users to access Internet information using devices with limited display space and bandwidth connection. The main problem is that such information often comes in formats that are designed for desktop environment with broadband connection, therefore special processing may be required before it can be displayed or output to a mobile device. The Wireless Application Protocol (WAP) is an attempt to solve the problem [WAP98]. It has a model similar to the WWW model in which Internet information can be downloaded into mobile devices. It has its own markup lan-

guage, called Wireless Markup Language (WML) [WAP00, HTK98], which is equivalent to HTML in WWW. The WAP architecture even provides a proxy-based mechanism to ensure interoperability with WWW. For example, HTML documents can be translated to WML format using a special filter, then be sent to a micro browser in a device in an encoded binary format to conserve bandwidth. WML itself uses a more structured approach in displaying documents (e.g., using the *card* and *deck* metaphors) which allows document organisation into smaller chunks.

To conclude this section, from the discussion in Section 2.3.2, it is clear that the wish to include mobility as part of application computation has a broad range of consequences that have to be taken care of by the application and its supporting system. In the development of such an application, appropriate software-oriented strategies have to be developed to handle mobility-related issues. The problem in implementing this idea is that it is difficult to create a universal problem-solving model due to the diversity of the issues to be supported. The discussion on how to deal with this problem is presented in the next section.

2.4 Handling of Mobility in Applications

There are so many approaches and strategies to handle mobility. This easily ends up with many types of applications. To provide a better support, it is necessary to focus on only a specific type of application. The criterion which is used to categorise the applications looks into whether computation that handles the mobility-related issues becomes an integral part of an application's main computation or not. This criteria is graphically shown in Figure 2.2. Only applications that handle mobility in their main computation will be considered in this thesis. From this point, the term *mobile application* refers to this type of application.

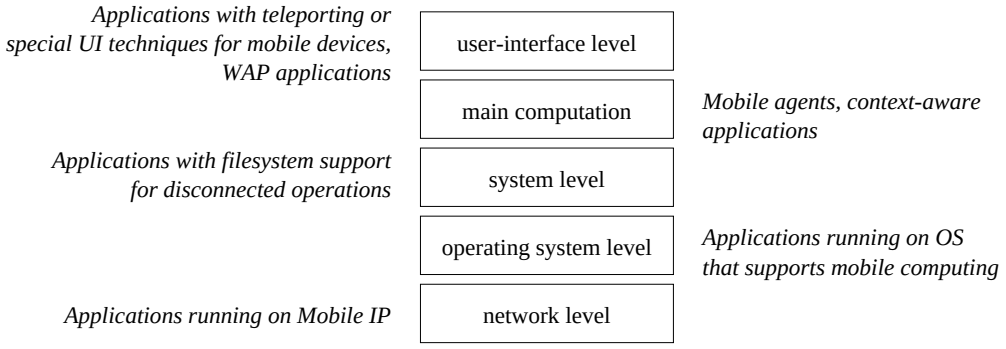

Figure 2.2: Handling of mobility at different level of computation

There are two types of mobile applications: those that intrinsically support user mobility and those that exhibit mobile behaviour through code migration. The first type is called *user-oriented* mobile applications, while the latter is called *code-oriented* mobile applications. This section looks at how mobility is specified and controlled in mobile applications. Descriptions are organised by migration principal (i.e., a user or a piece of code).

2.4.1 User-Oriented Mobility

Provision of mobility support in user-oriented mobile applications has two goals. Firstly, it enables an application to acquire information required to make an itself aware of the environment. Secondly, the awareness scheme can be pushed further to accommodate an adaptation strategy, which is required to minimise the negative effects due to the user's mobility.

To provide environment awareness and adaptability to a mobile application, a system that monitors the physical environment is needed. Contextual information is captured and passed to the application to be processed for either awareness or adaptation purpose. Environment monitoring can be approached by using a framework that defines the overall structure of a monitoring system and the way applications use it [BG98], or by employing a more loose system architecture using the *event* model. The latter approach

is gaining more acceptance (e.g., the OMG's Event Service [Obj00]) for the following reasons. First, it does not impose a tight coupling between applications and the underlying monitoring system. Additionally, it models the non-deterministic nature of the occurrence of environmental changes, and the asynchronous nature of the sender and receiver of an event.

An event-based monitoring system is mainly an architectural issue rather than a programming issue, so provision of its support is usually approached at the system level. For a system to be able to flexibly serve mobile applications, it must have the ability to process events from different types of event sources [WB98]. Additionally, it also has to build a new architectural layer for exporting the events to the higher level, due to the heterogenous nature of the event sources.

At the higher level, applications access a monitoring system using application programming interfaces (APIs). For example, the Odyssey system provides APIs to allow applications to monitor changes in some environment parameters [NPS95]. The APIs provide an organisational mechanism to allow an application to choose which events it wants to listen to. For example, Odyssey uses a naming scheme to identify event sources. A more structured approach is to classify event sources based on their characteristic using the object-oriented model [WB98]. Due to efficiency reasons, event notification is usually performed asynchronously using callback mechanism. At the application side, lazy checking techniques such as *futures* [Cha89] can be used to capture a callback without having to allocate too many computing resources.

From a programming perspective, implementation of awareness and adaptation of a mobile application is much the same as implementation of any other functional aspects of the application. It means existing programming languages and tools can be used as normal. If extensions have to be made (e.g., to embed a reasoning mechanism), they can still be done in the development framework offered by a programming language. This is the rea-

son why there are very few work on programming-specific aspects in this area.

2.4.2 Code-Oriented Mobility

Supporting code migration starts with building execution engine components that have to be distributed in network nodes. An execution engine for applications with mobile code is implemented by a run-time system provided by the development environment. Once a distribution of execution engines are available, it is possible to build a migration mechanism on top of them. In development tools based on a particular programming language (e.g., [ARS97, LO98, Obj99, GKCR97]), such a mechanism is built at the language level by incorporating it as part of the linguistic features offered by the language. Consequently, migration is geared to the underlying programming paradigm adopted by the language.

The reason for sending code to a remote location is to make a functionality available in that location. Migration of functionality allows functional code to be sent to a remote location and be executed there. At the language level, this has to be implemented using the language's constructs that capture the abstraction of a functionality. A functionality can be represented using a linguistic construct that exhibits the semantics of actions, including *procedures* or *functions* in procedural languages and *objects* in object-based or object-oriented languages. Object-oriented approach is more popular because it offers a more natural and structured way of approaching a problem [Mey97].

In the context of distributed computing, migration of procedural code can be thought as an extension of the remote procedure call (RPC) mechanism [BN84]. In RPC, a client makes a request of procedure execution to a remote server that owns that procedure. This idea is extended by *remote*

evaluation (REV), a client-server technology that allows a client to actively supply a remote server with executable code [Sta86]. The relationship between a client and a server is defined by a set of services, i.e., a collection of operations that a server can perform for a client. A programmer can then define client procedures that conform to the services, which can be relocated to a particular server that offers the services to be executed. This is done transparently through linguistic objects that refer to the server location. Remote evaluation is a generic technology; although it has been demonstrated in CLU [SG90], it can be implemented in other programming languages as well.

In more recent research work, code mobility is used to implement transportable agent systems based on the Tcl language [Ous94]. Tcl is an interpreted scripting language and has been ported to Windows and many flavours of Unix and Linux. A straightforward programming model for mobile agents is adopted by Agent Tcl [GKCR97]. An agent can be thought of as a manager for the creation and transportation of Tcl scripts that need to be executed in remote machines. It takes the form of a Tcl script that runs on top of a modified Tcl run-time system. An agent can create a child agent which can communicate directly with its parent. It can also be dispatched to a remote machine, and this feature can be used to migrate ordinary Tcl procedures.

A hierarchical approach to sending code is pursued by TACOMA [JvRS95]. Also based on Tcl, TACOMA uses the metaphor of folders, briefcases, and cabinets to represent mobile code and data, mobile containers, and static (non-movable) repositories, respectively. Similar to Agent Tcl, the term *agent* in TACOMA refers to a functionality of a code migration mechanism. An agent has control over folders, briefcases, and cabinets. Migration is characterised by a meeting of two agents from different locations. To migrate a Tcl procedure, the source agent packs the procedure and its

data into folders, then moves them to the destination using a briefcase. In the destination, the source agent meets the receiver agent which will unpack the briefcase and execute the procedure.

In the early part of the last decade, object-oriented languages started to gain popularity, mainly through C++ [Str97] and Java [AG97]. In an object-oriented environment, objects have the capability to represent real-world entities. They have states and behaviour, and become good candidates for representing mobile functionality. Another positive factor is a strong support for distributed computing by object-oriented languages, Java in particular. The inclusion of RMI [Mic98] and network-oriented libraries into Java's development kit [Sun99] makes it ready for programming of distributed applications. The combination of the programming paradigm and the language's rich features is the main reason why recent development tools for mobile code applications use Java as the underlying language. All these tools use object as the unit of encapsulation in which migration mechanism is applied on.

To make an object mobile, it is necessary to assign a migratable semantics to the object. There are two approaches to achieve this. In the first approach, the semantics is embedded in a particular type (i.e., class), and user-defined objects obtain this semantics through object-oriented mechanism such as inheritance, i.e., a mobile object is created by subclassing a particular class. Examples of development systems that adopt this approach include Aglet [LO98], Mole [SBH97, PH98], and Ajanta [TKV+98]. A variation of this approach uses Java's *interface* mechanism to apply mobility. In Voyager [Obj99] for example, a mobile object can only be created from a class that implements special interfaces. In the other approach, a new construct with mobility semantics is specially introduced. The construct typically denotes a container with mobile capability. An object is made mobile by loading it into a container. This approach is taken by Sumatra [ARS97].

Object migration in Java-based systems is mainly implemented using the Remote Method Invocation (RMI) mechanism [Mic98]. With respect to the communication pattern, RMI can be thought of as the object-oriented equivalent of the remote procedure call (RPC) mechanism. The ability of calling remote methods is exploited and used to implement the migration mechanism. Development tools use this feature to build a new layer on top of RMI, which encapsulates the details and provides a high-level, migration-oriented abstraction, such as the notion of location (e.g., *cells* in Hive [MGR+99] or *agent servers* in Ajanta [TKV+98]).

All development systems so far discussed in this section have some similarities in their approach to providing code mobility. They preserve the original syntax and semantics of the underlying language, and migration mechanism is implemented as extensions to the linguistic framework. There are other development systems that take a different approach. In these systems, the distribution semantics is *directly integrated* into the linguistic framework. This makes the languages naturally possess the ability to cleanly express migration-related issues.

Obliq is an object-based language whose lexical scoping also determines the distribution of a program's computation elements [Car95]. The distribution semantics of Obliq is based on the concepts of *sites* (i.e., address spaces), *locations*, and *values*. These concepts are assigned to Obliq's linguistic constructs, for instance, variable identifiers denote locations, constant identifiers denote values, while sites can be implicit in the creation of a location or explicitly stated by *execution engine*. Data and procedures can migrate from one site to another, and the meaning of distributed computation is determined by the *bindings* of locations to sites, instead of by execution sites themselves. Data and procedure migrations are also transparent; no explicit command is required. The language has been used to demonstrate the ability to migrate a whole application, including its user-interface [BC97].

An approach similar to Obliq's is taken by Distributed Oz [RHB+97]. The language is object-oriented and has a set of linguistic constructs whose semantics is divided into *language* semantics and *distributed* semantics. The language semantics defines the linguistic meaning of the constructs, while the distributed semantics defines their behaviour upon the application of distributed operations. To program a mobile application in Distributed Oz, one has to determine the degree of mobility of objects in order to use the appropriate constructs.

2.5 Mobile Application Development From A Software Engineering Perspective

The previous section in this chapter presented a survey on system and architectural support for mobile computing and how it is programmed into mobile applications. This section carries on to the development aspect of mobile applications. Modeling is specifically discussed because it works on system-level specifications which become the embryo of mobile applications. This section also discusses some issues on the implementation of mobility support.

2.5.1 Modeling and Specification of Mobile Systems

Researchers have been trying to understand mobility as a computing phenomenon for quite some time. There are calculi that have been proposed to model and describe mobile systems. However, early work on these proposals do not depart from the mobility point of view, since the notion of mobile computing has not been widely known during that time. For example, the π-calculus is a process calculus that allows a channel between processes to move along other channels [MPW92]. The calculus is designed for concurrent processes, but it can be used to capture the notion of process migration

through channel movement. However, as movement is applied on channels instead of processes, process movement as a phenomenon cannot be distinctively indicated.

The π-calculus is extended to have better locality using the *chemical abstract machine* (CHAM) concept [BB92]. CHAM is a framework for specifying reductions which uses the notions of chemical reaction in a solution and membranes separating subsolutions. The framework has been used to introduce the notion of location to processes [FGL+96]. A location resides on a physical site and contains a group of processes. Mobility is regarded as a primitive that can atomically move a location to a different site.

Mobile Ambient [CG98] is a calculus based on the concept of bounded space (i.e., *ambient*). Ambients can be used to represent process localities. They can also be structured to form a hierarchy. This is achieved through operations on ambients, called *capabilities*. The use of ambients to represent locations is intuitive, because they model physical properties of a computation locality (e.g., the bounded nature of the locality, as well as the capabilities to enter, exit, and open a locality). Accordingly, mobility can intuitively be represented by the dynamics occuring within ambients. In Mobile Ambient, both processes and ambients can be migrated by applying capabilities.

A different approach to modeling mobility is taken by Mobile UNITY [RMP97]. It is based on UNITY [CM88], a state-based, reactive model for concurrent and asynchronous systems which uses a specialisation of temporal logic. In UNITY, a system's behaviour can be reasoned through its safety properties (i.e., those which define allowable states in a program) and progress properties (i.e., those which define the operational semantics of a state transition). Mobile UNITY extends this feature to facilitate specification of mobile systems by introducing new concepts to deal with movements and interactions between mobile entities.

UNITY also comes with a set of programming notation for specification purposes. System specification can be written in a program structure similar to that of Pascal. System properties are specified using program statements. A UNITY program does not have an explicit execution model. To describe the reactivity of a system, an *artificial* execution model is devised. In this scheme, each statement is executed infinitely often in an infinite time period. Mobile UNITY uses UNITY's framework for expressing a system. It introduces new constructs to the framework to facilitate its mobility-specific concepts.

The modeling frameworks that have just been mentioned focus on the formal aspect of model representation. The purpose of such a framework is to set a foundation for a computation system in mobile computing. Foundational modeling systems aim to define a complete computation system using their proposed abstractions. Therefore mathematical formalism is normally used to represent a model. Proof of correctness of a concept or theorem has more emphasis than the selection of model abstractions that can aid the development of a mobile application. The latter aspect is essential if a model is to be used to specify a set of software requirements during the analysis stage of an application development process.

Requirement analysis is a software engineering task that enables developers to specify software functions and performance, indicate software's interface with other system elements, and define constraints that must be met by the program being developed [Pre97]. It is a process of discovery, modeling, and specification that works on the software scope. Models of the data, functional, and behavioural domains of the program are defined. In this respect, the relevance of modeling of mobile systems becomes apparent, since it yields a model that specifically describes essential mobility-related aspects. The model is a valuable complement to other sources of information that is used to develop a requirement specification for the software.

When a model is used in the scope of mobile application development, its abstractions have to undergo a series of refinement process. This is not a simple process if foundational models are involved, since their abstractions are not designed for this purpose. They tend to leave out pragmatic semantics which leads to clear and unambiguous representation of abstractions in a refinement process. For instance, consider the abstraction of a *process* found in many theoretical modelling systems. Its implementation scenario may not be straightforwardly obvious, because it does not provide sufficient information for creating a software model of it (e.g., what functionality it has, and how it relates to other processes). In general, lack of pragmatism creates a wide abstraction gap between theoretical modeling and its implementation. A considerable refinement effort has to be put to bridge this gap. This makes such a process non-trivial.

In the software engineering area, researchers have contributed a number of modeling tools. Since they are designed from the software engineering point of view, they use abstractions that are easily implemented in a development project. For example, the Data Flow Diagram (DFD) is a modeling tool that emphasises on the flow of processes [Pre97]. However, the notion of processes in DFD is quite different to that adopted by foundational modeling tools. DFD processes are *software* processes, indicating processing activities that can be mapped directly into a software architecture. Another example is the Unified Modeling Language (UML) whose design is strongly related to the object-oriented paradigm [RJB99]. Object is a generic construct that could represent any real-world entity, but UML provides a development-oriented framework to work with objects so that it becomes easy to map a model into a software design.

The problem with these modeling tools is that they are not specifically designed to deal with mobile systems. A DFD can represent mobility as a process but it cannot clearly distinguish it from a functional process.

The case of UML is similar. UML consists of a set of *views* that can be used to model different aspects of a system. For instance, the physical aspect of a mobile system can be modeled using the physical view, and the relationship between a mobile entity and its environment can be described using the state-machine view. However, these views are designed with no specific consideration on the mobility aspect, therefore it is difficult to grasp a strong mobility abstraction from them. Description about mobility aspects are split into different views, preventing developers from having a unified perspective of them.

2.5.2 Implementation of Mobility Support

Development tools for mobile applications generally follow a framework that is usually based on a simple programming model. For example, most of Java-based development tools use the concept of *mobile objects* to model code mobility (e.g., [Obj99, ARS97, LO98]). A mobile object is an ordinary Java object that has the ability to move to another location. In user-oriented mobile applications, the same model can be used to implement awareness and adaptation mechanisms. As mentioned in Section 2.4.1, implementation of such a mechanism is much the same as implementation of any other functional aspect of an application. In general, the object-based programming model has gained wide acceptance for its simplicity.

With the increasing demand for more comprehensive mobility support, the problem is not on the concept of object as an abstraction representation. Instead, it lies in the object-orientation mechanism that is used to *compose* objects. In a programming activity, programmers should strive for well-modularised software components, and generally object-orientation is well-suited for this purpose. It supports the ideas of separating programming concerns [Par72] and localising them into software modules with well-defined interfaces. The concept of "objects as modules" fits into this principle, and

this becomes the reason why object-oriented languages are widely used in solving complex problems. Recent research findings, however, show that it cannot always cope with programs with non-functional components.

It has been shown that object-orientation exhibits some incompatibility with synchronisation [MY93, NS99, CG97] and mobility [NS00] mechanisms embedded in object-oriented programs. The original idea of separation of concerns that forms the basis for the concept of modularity is not capable of handling multiple concerns in *different* domains (i.e., as opposed to the single, functional domain). Researchers have come up with proposals that extend the idea of separation of concerns to cover multiple domains [KLM+97, TOHJ99, Ber94]. With a suitable programming toolkit (e.g., AspectJ [LK98] or HyperJ [TO00]), it is possible to separate functional components from other components that represent other concerns (i.e., cross-domain composition). The idea is to allow composition of software modules implementing different concerns without having to sacrifice the interest of any concern.

The improved concept of separation of concerns, however, has not been well tested and applied in mobile application development. One inherent obstacle is the tight coupling between component composition and the technology to implement the mobility support. In Java, for example, code migration mechanism can be built on top of RMI technology, but seamless integration of RMI and object-orientation which satisfies the principle of separation of concerns is difficult to achieve. This is because RMI programming requires explicit low-level initialisation steps before remote communication can be performed [Mic98]. Programming tools such as Voyager [Obj99] use high-level abstractions (e.g., interfaces to mobility-related features) to wrap the low-level details, but this approach does not completely solve the problem because mobility abstraction is still mixed with functional abstraction. The compositional burden may even be worse when environment awareness

and adaptation is involved, because the mechanism may have to be migrated along with the migrating code as well.

Concentration on the programming aspect in providing mobility support describes a bottom-up strategy in supporting mobile applications. In this approach, such support is not backed up by strong conceptual modeling. Instead, it grows naturally from the programming level. This phenomenon is understandable since mobile computing is a logical extension of distributed computing, an area where programming support has started to mature.

The bottom-up strategy has a drawback. It cannot provide a comprehensive solution due to the lack of a strong conceptual basis. Since mobility modeling is not specifically facilitated during the requirement analysis stage, developers are forced to use a simple mobility programming model created as an extension of certain programming concepts. However, the extension is *ad-hoc*, since it is made to suit a particular type of problem, indicated by the presence of a dominant type of principal of migration. When the problem expands and it cannot be handled by the default model, developers have two options. They can either modify the software requirements to fit into the new problem description, or perform workarounds by introducing additional abstractions to fill the gap. The first option incurs additional time, effort, and cost, and often leads to an expensive software development project. The second option tends to introduce complexity to the composition of the resulting software.

To summarise the discussion on the software engineering aspect of mobile application development, it can be asserted that there is a discontinuality in mobile application development process. At one end, theoretical work has produced some modeling frameworks to create models that describe the generic aspect of mobile computing. However, the abstract nature of the models necessitates the insertion of an intermediate layer that allows easy transformation to a realistic implementation. At the other end, current

programming technology still cannot provide a comprehensive support for implementation of mobility support in mobile applications.

2.6 Research Issues Revisited

The previous sections of this chapter presented a survey of research on the mobile computing area, especially those which have relevance to the development aspect of mobile applications. This section concludes this chapter by revisiting the research areas given in Section 1.5 and provides justifications for their relevance.

2.6.1 Modeling of Mobile Systems

The first issue to be addressed in the research is the abstraction level in which provision of support for mobility should be initialised. The lesson learned from current programming tools suggests that it is no longer sufficient to think of mobility as a dependent attribute of a principal of migration (e.g., an object, a user, or a piece of code). Thinking of mobility as an attribute makes it an inseparable part of a mobile entity, and therefore cannot be exploited independently of the entity's requirements. As a result, support for mobility can only be narrowly provided in the framework of the mobile entity's main agenda.

A narrow perspective on mobility would not give satisfactory solutions if applied in situations where multiple mobile entities with different mobility requirements are present in the same program. Supporting the requirements is indeed possible, but this has to be done using a bottom-up approach (i.e., support is constructed from the primitives offered by the programming language). This approach is not only inefficient, but also prevents developers from systematically viewing the overall situation. The latter case is extremely important if a generic and uniform approach is sought.

It can be hypothesised that the other way of approach (i.e., the top-down manner) can give better result. The approach starts with a generic abstraction of mobility in which all its essential features can be captured into a model. To fulfil its role, a model for mobile systems has to meet several criteria, described as follows.

Genericity

Genericity is a common feature of any modeling system. In this research, genericity is translated as the ability to provide uniform support for different situations commonly found in mobile applications, mentioned as follows.

- *Mobile applications with different mobility support requirements.* Different types of principals of migration require different mobility support. User mobility requires applications to be context-aware and context-adaptive. Code mobility requires programmers to define the parts of the application that need to be migrated to different locations, as well as to specify the migration destinations.

- *Mobile applications with multiple contexts.* First of all, a model has to be able to capture the concept of context. Furthermore, multiple contexts may be present at the same time in a mobile application if the performance of the application is sensitive to different environmental elements represented by the contexts.

- *Implementation of a model specification on different computing environments.* Mobile applications can be developed in different implementation environments. A development environment can be selected based on the type of support provided by an application (e.g., the target principals of migration), or the availability of the development tools.

To achieve genericity, a model should be able to capture the notion of mobility as an independent phenomenon, rather than a dependent attribute of an entity. Capturing mobility as an independent concept allows programmers to handle its aspects as a subject of manipulation. It means their existence and behaviour can be defined independently, irrespective of the type of the entity and the development environment.

Separation of Concerns

The main motivation to promote separation of mobility from functionality is that mobility has its own programming domain which is separated from that of functionality. Under this perspective, it should be possible to conduct development of one aspect independently of the other. In particular, it should be able to implement mobility support using abstractions from its own domain. Also, early awareness of the importance of separation of concerns leads to careful composition of program components. This help programmers achieve better program composition at the implementation level.

In fact, the principle of separation of concerns is used as the driving force to achieve genericity. Genericity can be achieved if a universal abstraction of mobile entities can be provided. Separation of mobility from functionality helps achieve this goal by drawing a clear boundary between the two aspects, allowing their specification and implementation to be carried out independently as well.

Pragmatic Representation

A modeling framework should be complete, in the sense that it should be able to describe any mobile system. But to be usable in a development process, its pragmatic aspect should be highlighted as well. A model generated from the framework should minimise the *abstraction gap* between itself and the

tool used in the subsequent development stage. In other words, it should be able to capture essential features of mobile systems, but at the same time contain necessary information to guide its implementation.

2.6.2 Specification of Mobile Systems

As mentioned in Section 2.5.1, the main difficulty of using a theoretical model as a basis for mobile application development is the incapability of the model representation to pass meaningful information regarding its abstractions to the development stages beyond modeling for further refinement. This situation prevents a smooth abstraction refinement process, since a developer has to supply the required information during the process. Another reason not to recommend this approach is that it allows arbitrary abstraction interpretation by a developer. This may lead to imprecise or even incorrect results.

The research addresses this problem by focusing on the representaion of the model. A model representation should provide sufficient information to guide its implementation. It should expose a *development-oriented* representation of a system, on which further development stages can be based. It should stress the ability to deliver its expressive power, rather than act as a proof system. A *specification language* is used for this purpose. The reasons why a language is selected as a representation tool are listed in the following.

- It is relatively easy to design a language with sufficient *expressive power* required to represent a model.
- A language can be designed to operate in an abstract level without losing its expressive power.
- From a programmer's point of view, a linguistic representation is psychologically more acceptable than a mathematical one.

The specification language is used to carry out *mobility specification*, that is, specification of mobility control (i.e., the properties and behaviour related to mobility) of a mobile application. This denotes another novel feature of the proposed development approach. Unlike the traditional approach which does such a specification at the programming level (hence the term *mobility programming*), mobility specification is conducted at a more abstract level. Mobility control is specified using linguistic constructs with abstract semantics which need to be implemented in a later development stage.

Moving mobility specification to an abstract level has some interesting consequences. First, the specification language transforms a modeling activity into a specification activity. The term *pragmatic modeling* (as opposed to *foundational modeling*) would probably fit to illustrate the situation. Modeling is carried out in terms of pragmatic expressions, while the abstract nature of the system is still preserved.

Secondly, mobility specification conforms to the principle of separation of concerns. It promotes mobility to a higher abstraction level that form an isolated development domain, separated from the one that is used to develop the functionality of a mobile application.

Lastly, specifying mobility control with some degree of abstractness is preferred as it preserves the genericity offered by the model. As previously mentioned, this allows flexible implementation of the model. Possible cases include the use different programming languages, the flexibility to select certain implementation options (e.g., whether to implement all features of the model or not), and implementation on a non-programming environment.

2.6.3 Implementation of Mobility Support

The implementation part of the research evaluates the usability of the modeling framework and the specification language in an abstraction refinement

process. Given a model, how easy it is to develop a program that implements the requirements specified by the model. The research focus is not only on the outcome, but also on the lessons learned from the activity. The research also analyses the fitness of the implementation environment for a model implementation.

Two prototypes are developed. The first prototype follows the programming course where linguistic constructs of the specification language are translated into equivalent Java programming constructs. A programming framework that help programmers conduct a model implementation is developed. The selection of Java as the target language is made on practical reasons. Its object-orientation is a well-known paradigm for developing complex applications. Moreover, it is well-supported by various tools that greatly help programming of network-oriented applications.

The second prototype is a cron-like application running on a Linux shell environment that is designed to allow mobile users to organise command execution based on their locations. The idea of this work is to show that it is possible to extract essential concepts from a model and implement them in a non-programming environment (i.e., Linux environment in this case) using available tools provided by the environment (i.e., tools included in a Linux distribution).

Having presented the research background and stated the research problems in this chapter, the subsequent chapters describe the findings and contributions of the research. The description starts with the proposed modeling framework for mobile application development.

Chapter 3

Mocha: A Modeling Framework for Mobile Application Development

3.1 Introduction

This chapter marks the beginning of a series of chapters explaining the findings of the research reported in this thesis. As stated in Section 2.6.1, the effort to support mobile application development through a conceptual approach necessitates the exploration of modeling of mobility. In particular, the exploration should focus on how generic properties and the behaviour of a mobile system can be represented in a model.

Traditionally, the notion of mobility modeling in a mobile application development process cannot be clearly identified. Developers specify mobility control at the programming level using the default, general-purpose programming model supplied by the language. Section 2.5.2 explained that this traditional approach as shown by current development tools are not sufficient to satisfy the need for a comprehensive solution, so a more conceptual approach should be pursued.

The main motivation behind the proposed approach is the need to represent the unique aspects of mobile systems. Preservation of execution support, fluctuation in resource availability, and penetration of environment elements into computation are the primary issues that are present in every mobile application. They cannot be addressed comprehensively at the programming level, therefore it is necessary to shift the focus of the approach to a more conceptual level.

In mobile application development, modeling becomes the first development stage since it describes mobility-related requirements that have to be satisfied by the application being developed. Therefore the design of the modeling framework should reflect the close association between a model and its implementation. In addition to the *completeness* requirement (i.e., it allows modeling of any mobile system with any kind of mobility requirements), it has to consider the *implementation factor* (i.e., it has to use concepts that can be brought along to the implementation phase and implemented without much difficulty and loss of semantics).

It is not sufficient for a generated model to use abstract concepts such as processes and channels [MPW92] or ambients [Car99] because they do not bear sufficiently specific information to be refined in the development stages. Using such concepts in the model will place the burden of refining the abstraction of the concepts in the design and implementation stages. This is not desirable since it may potentially create unnecessary complexity at these stages of development.

Using a single concept to identify the unique aspect of mobility, such as location and bounded space [CG98, TA98], as the basis for modeling is not sufficient either. As discussed in Section 1.4, location is not the only source of dynamics that has to be handled in a mobile application. In fact, mobility can be expressed without using the notion of location at all. This assertion is based on an assumption of *indirect location representation*, where

the dynamics within an environmental element can represent change of locations. For example, a change of neighbourhood (i.e., people logged in the same machine) may suggest a change of the user's location.

The proposed modeling framework is designed with the previous considerations in mind. The framework, which is called **Mocha**[1] is described in this chapter. A preliminary work on Mocha is described in [NLS+00]. Mocha is designed to intuitively specify a mobile system. It is based on an extension of the UNITY specification system [CM88] which is designed to work with reasoning of mobile systems [WR96], and Mobile Ambients calculus [CG98]. A brief introduction to these modeling systems are provided in Section 3.2 as a background. The description of Mocha starts from Section 3.3 that identifies important issues that have to be addressed by Mocha. Section 3.4 describes the formalism of Mocha, and this is followed by Section 3.5 that explains how different types of mobile applications can be modeled using Mocha. Finally a discussion on the features of Mocha is presented in Section 3.6 to conclude this chapter.

3.2 A Brief Description of the UNITY Model and the Mobile Ambients Calculus

3.2.1 The UNITY Model

UNITY is a computational model and a proof system for program specification [CM88]. The main feature of UNITY is that it separates program specification, which centers around the proof of correctness of a program, and its *mapping* to a particular target architecture. In other words, UNITY separates the *what* aspect of a program (i.e., the specification) from the *how*

[1]The word Mocha means flavoured with coffee and chocolate. This name was originally selected for its close association with Java to name a byproduct of the research, a set of extensions to Java. As the research expanded, the name is now used to refer to both the modeling framework and the specification language.

(i.e., the algorithm), *where* (i.e., what computer architecture it is going to be executed), and *when* (i.e., when a program statement should be executed, in terms of processor execution sequence on a particular architecture).

The UNITY modeling system uses a formalism which is a specialisation of linear temporal logic. The formalism is based on assertions of the form $\{p\}s\{q\}$. A statement of this format means that an execution of an action s in any state that fulfils predicate p will result in a state that satisfies predicate q. This format is used to specify *properties* of a program, which are categorised as *safety* or *progress* properties. Safety properties define allowable states that a program can have. Progress properties define the operational semantics of a transition for a given state. The formal specification of a UNITY program can be specified using the safety and progress properties of the program, written as a collection of logic statements.

UNITY introduces some predicate relations as part of its logic system. They are **unless**, **ensures**, and **leads-to** (denoted by the "$\mapsto$" symbol), and they can be used to derive further relations in UNITY's logic statements. These relations are described in Table 3.1.

Relation	**Remarks**
p **unless** q	If p holds when q does not, it remains so as long as q does not
stable p	Defined as p **unless** $false$
inv p	Defined as (**INIT** $\rightarrow p$) $\wedge$ (**stable** p)
$p \mapsto q$	If p holds, then eventually q will hold. It does not matter whether p still holds when q becomes true
p **until** q	Defined as $(p \mapsto q) \wedge (p$ **unless** $q)$
p **ensures** q	This relation holds if p **unless** q holds; and there is an action that, if executed while $(p \wedge \neg q)$ is true, makes q true; and there is a guarantee that the action will eventually take place.

Table 3.1: UNITY's logic relations

UNITY also introduces a three-part constructor statement which can be written using the following syntax.

$$\langle \textbf{op} \ \ variables \ : \ range\_constraints \ :: \ statement\_expr \ \rangle \qquad (3.1)$$

The **op** can be a binary, associative, or commutative operator, such as Σ (cumulative addition), Π (cumulative multiplication), $\forall$ (universal quantifier), and $\exists$ (existential quantifier). Logically the constructor creates a multi-set of values $(v_1, v_2, \ldots, v_n)$ by evaluating the statement expression for every possible instantiation of the variables that fulfil the range constraints. The constructor gets its final value by appropriately evaluating the overall expression according to the given operator.

Wilcox [WR96] extends UNITY to work in a mobile system environment. Specifically, he introduces the concept of place, time, and action into UNITY. The notion of place is introduced by attaching location as an additional attribute in state representation. This is done using the **at** λ *spatial notation* where λ refers to a location.

The notion of time is introduced because temporal properties are closely associated with mobility. For example, when the system designer is interested in the time-bound behaviour of mobile elements, mobility is intrinsically bound to the passage of time. The timing system is event-based; a *timestamp* is a sequence of time values representing the successive, individual occurrence of an event denoting a transition from false to true for a given predicate. Temporal properties of a system can be specified as *timing constraints*, which are expressed in terms of timestamps.

The UNITY extension also provides an abstraction of actions which can be associated with individual mobile elements. In the UNITY framework, an action can exist without having to be enabled (i.e., when the state does not allow it to be executed). An action is tied to a location, and is encapsulated in the **takes-to** relation. The expression

$$A \textbf{ at } \lambda \textbf{ takes } P \textbf{ to } Q \textbf{ when } R \tag{3.2}$$

means that action A, if executed at λ, will transform P to Q upon the satisfaction of predicate R.

3.2.2 Mobile Ambients Calculus

Mobile Ambient is a calculus for mobile computation that is based on the concept of an *ambient* or bounded place [CG98]. An ambient is usually used to refer to a physical entity as commonly found in real-life mobile computing (e.g., a network node, which is bounded by its physical structure), but the same notion can be extended to cover unconventional instances like a web page (bounded by the file) or an Internet address space (bounded by a range of IP address). An ambient has structural characteristic, in the sense that it can contain one or more ambients, forming a hierarchy. An ambient can also move as a whole by carrying its contents. Using this concept, mobile computation can be considered as moving a computation (possibly with its surrounding ambient) from an ambient to another ambient, crossing their respective boundaries.

The syntax of the calculus is described as follows. It has *processes* (indicated by P and Q) and *capabilities* (indicated by M) as the main components.

$$\begin{array}{lcrl} P, Q & ::= & \mathbf{0} & \text{inactivity} \\ & | & (\nu n)P & \text{restriction} \\ & | & P|Q & \text{composition} \\ & | & !P & \text{replication} \\ & | & n[P] & \text{ambient} \end{array} \tag{3.3}$$

$$
\begin{array}{rcrl}
 & | & M.P & \text{action} \\
 & | & n & \text{names} \\
M & ::= & in\ n & \text{can enter } n \\
 & | & out\ n & \text{can exit } n \\
 & | & open\ n & \text{can open } n
\end{array}
$$

In an ambient $n[P]$, n is the name of the ambient where a process P is running inside it. The execution of P is independent of any other operations, i.e., P can be actively running in parallel with other processes and operations, including those which operate on the host ambient.

Capabilities define actions or operations that can be performed on ambients. The notation $M.P$ represents an action defined by a capability M, and then proceeds with a process P. The behaviour is synchronous, which means P will not start until the action pertained to the capability is executed.

There are three types of capabilities: one for entering an ambient, one for exiting an ambient, and one for opening an ambient. In an action $in\ m.P$, the entry capability $in\ m$ tells the ambient surrounding the action $in\ m.P$ to enter a sibling ambient m. If m does not exist when the operation is executed, it will block until such a sibling exists. The behaviour of this capability is shown by a *reduction* (i.e., atomic computation step) as follows.

$$n[in\ m.P \mid Q] \mid m[R] \rightarrow m[n[P \mid Q] \mid R]$$

The ambient n, which contains processes P and Q, is a sibling of another ambient m, which has a process R. The action $in\ m.P$, if executed successfully, transforms n as the child of m, followed by P. Both P and Q also move in accordingly.

An exit capability $out\ m$ in an action $out\ m.P$ would cause the ambient surrounding the action to exit the parent ambient named m. If there

is no such parent, then it blocks until such a parent exists. The following reduction describes the operational semantics of the exit capability.

$$m[n[out\ m.P \mid Q] \mid R] \rightarrow n[P \mid Q] \mid m[R]$$

The exit capability is the opposite of the entry capability. In the above reduction, if the operation is successful, it moves out an ambient n from its parent m, followed by P. Both P and Q are transformed to the higher level hierarchy of ambients.

The final capability is opening an ambient, *open* n. If used in an action *open* $n.P$, it will dissolve the boundary of n located at the same level of the action. If n has children, the children become unbounded from n. This is shown by the following reduction. If no n is found, the operation blocks until it finds one.

$$open\ n.P \mid n[Q] \rightarrow P \mid Q$$

3.3 Key Issues in Mobile Applications

This section discusses several important issues in mobile applications. An informal scenario is first presented to illustrate situations commonly experienced by mobile applications. This is followed by detailed discussions about the concepts presented in the scenario. This section provides a rationale for the proposed modeling framework, as the concepts presented in this section will be used as major ingredients for the design of the framework.

3.3.1 A Scenario of a Mobile System

The scenario uses an imaginary background of futuristic life where humans have already been able to conquer the outer space[2]. In this set up, scientific

[2]The theme is inspired by sci-fi movies like *Star Wars* and the *Alien* series.

research stations have been built on several planets. These stations host machines and computers which can be activated and controlled remotely. The stations also function as docking terminals where space ships can land, and as communication stations in which communication signals between space ships can be relayed and propagated.

In an interplanetary mission, the goal is to perform a set of chemical analysis of some specific samples collected from different planets. The mission is conducted by an astronaut in a space ship that is also used as the vehicle. The mission may have to make use of computers in remote stations for analysing the data. These computers are fully controlled by the astronaut, and can be activated by sending control programs via communication links.

In collecting the samples from a particular planet, the astronaut lands the space ship in a planet, then sends a robot to explore the planet's terrain. The robot is programmed to find the samples and return to the space ship after collecting them. The astronaut does some lab tests, and probably needs to send the data to remote computers for further analysis, and get the results back for further processing. After finishing with one planet, the astronaut moves to the next destination planet.

Meanwhile, the space ship is equipped with some form of intelligence to avoid obstacles in the outer space that can fail the mission. The intelligence is implemented as an autopilot feature that can detect atmospheric disturbances during its journey (e.g., meteor showers). The ship is equipped with control mechanisms to make adjustments should such problems occur. For example, it can use a different route or change its cruising speed.

It should be noted, although the scenario uses a fictional theme, it can precisely describe real-world cases which may be found in a near future. For example, the astronaut could be an ordinary mobile user who owns an

intelligent car (the space ship). The car is equipped with a portable computer constantly connected to the Internet, so the user is able to send mobile code (the robot) to, for instance, some other computers to retrieve some documents (the samples). The car's computer is also capable of selecting the best route and adjusting the car's speed (the autopilot) based on a given destination, its current location, timing constraints, and traffic density.

3.3.2 Functionality vs Mobility

Inherent in any mobile system is the capability of performing an action, which is called *functionality*. Functionality is shown in the scenario by the astronaut that perform certain tasks, the remote computers that does computing analysis, the space ship controller that perform adjustments required during the trip, and the robot program that performs data sampling.

Functionality encapsulates the capability of executing an action into a single abstraction. A functional entity is one that bears a function that can be executed somewhere sometime, and these spatial and time points can be reached using its ability to move. The definition does not distinguish between the types of actual entities. Code, data, and human share the same property and therefore they are treated uniformly as functional entities by the definition. Of course there are semantic differences between different types of entities (e.g., "functional code" and "functional human" have different meaning), but they are transparent at the model level.

The notion of being able to move, or the *mobility*, of an entity is shown in the scenario by the spaceship and the robots. It is denoted by a *vehicle*, borrowed from the transportation metaphor, to denote a construct that has an intrinsic capability of moving. To have a vehicle is like to be able to move from one place to another. Assigning a vehicle to a functional entity means embedding its mobile property to the entity.

It is arguable whether mobility is a different form of functionality. This view is based on the fact that mobility itself implies the ability of doing something. In fact this view is shared by current mobile application development tools, where migration features are implemented using constructs normally used to implement functionality, for example as class methods in Voyager [Obj99] and Aglet [LO98]. However, Mocha looks at them as distinct features. In Mocha, mobility and functionality are orthogonal to each other. Being able to do some action is considered to be different from being able to move. Functionality is an intrinsic feature of a Mocha entity, while mobility property can be dynamically assigned to or removed from it.

The relationship between functionality and mobility is characterised by a *structure* created from a hierarchy of functional and mobile entities. A structure is constructed when a functional entity is loaded into a vehicle, creating a mobile functional entity (or mobile entity in short). When being contained in a vehicle, a functional entity is movable along with the vehicle. This is shown, for example, by the astronaut that rides the space ship. Containment construction is the mechanism used by Mocha to assign mobility to a functional entity. In other words, a mobile entity is always composed by a vehicle and one or more functional entities.

A relationship between a functional entity and a vehicle is dynamic. A vehicle can drop off its contents as well as load them in. When an entity is unloaded from a vehicle, it loses its mobility.

A vehicle can also load other vehicles. A hierarchy of multiple vehicles can go to any level of depth, and each vehicle can have their own functional entities. This situation is commonly found in systems with multiple mobility agenda.

A vehicle containment also defines the scope of *mobility properties* of a vehicle. A mobility property is a property that in some ways define

the behaviour of a mobile entity when it is in effect. Examples of mobility properties include patterns of movement (e.g., the itinerary of the space ship in the scenario), migration constraints (e.g., the space ship can land on terrestrial stations but not at space stations), and adaptation features (e.g., the autopilot feature of the space ship). All entities inside a vehicle are subject to the vehicle's mobility properties, as long as they stay in the containment of the vehicle.

One of the mostly used mobility properties is patterns of movement. Such a pattern is expressed in terms of abstractions referring to a particular type of physical environment elements. In many cases, it can be pre-planned or computed before the migration starts. Mobility with this type of movement is called *deterministic mobility*. In some other cases, however, such a pattern cannot be computed. This situation occurs in mobile computing where a pattern of movement is not important, but the ability to execute a task in *any* location is. In this situation, mobility becomes *non-deterministic*. For example, in a cellullar phone application (e.g., WAP browser), the destinations of a user are non-deterministic and the pattern that shows where the user goes is not important for the application.

3.3.3 Separation of Mobility from Functionality

In designing a mobile application, it is crucial to identify the tasks required to solve a problem with mobility. Interestingly, such tasks can be decomposed into functional and mobility elements. Using the structuring mechanism of vehicle, it is possible to describe a task structure in terms of the mobility of the functional components. For example, Figure 3.1 shows the structure of the tasks in the space ship scenario.

Figure 3.1 shows that a mobile application can be decomposed into a hierarchy of problem-solving components. There are different tasks which

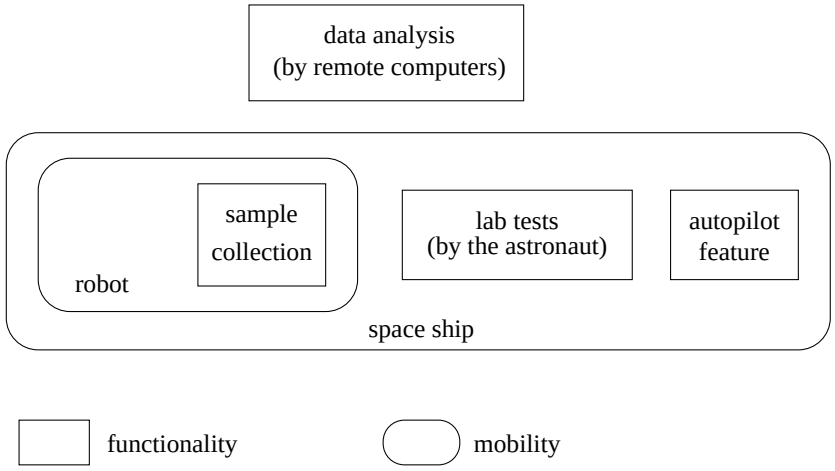

Figure 3.1: The task structuring of the space ship example

have to be done by migrating some functions. In each task component the mobility component sits at a higher level than the corresponding function. This shows that mobility has an abstraction level higher than that of functionality, and it becomes the observable characteristic of the task.

Task structuring is useful to identify functional components and their mobility property during a software analysis process. Such structuring can be achieved by first identifying the functions required to accomplish the overall task. This is followed by organising them based on their migration requirements. Functions with the same migration requirements are grouped into the same vehicle. Multi-level vehicle structures (as shown in Figure 3.1) can be used if there is a need for mobility with different agenda (e.g., different itineraries).

An important requirement for task structuring is the explicit separation of mobility from functionality. Doing such a separation at the early stage of software development pays a big advantage. To recall Section 2.5.2, it is easier to satisfy software modularisation criteria if the concerns are clearly separated out. Program composition can be made without having to sacrifice the interest of any concern. Separation of functional and mobility issues in

the first place allows categorisation of software components, which can be of assistance for further design process in aiming a modular design.

In Mocha, distinguishing mobility from functionality also assists the selection of the appropriate technique for a particular problem. It has been understood that computing with mobility may not be the only viable approach to problem-solving [CHK97]. For example, many problems solvable by mobile agents can also be approached using traditional distributed computing techniques (e.g., client-server). It is desirable to decouple the problem-solving activity from the technique used. To achieve this, one needs a design mechanism that allows flexible selection of the technique that will be used in the problem-solving.

Separation of mobility from functionality could help in this situation. Functionality is the core element of any problem-solving activity, irrespective of the problem-solving technique used. Mobility, on the other hand, is only one possible way, which can be substituted by other methods. This suggests that the presence of a function is invariant in the system's design, and its development can be decoupled and done independently of the decision of making it mobile. More specifically, there is no need to make an early decision of whether a functional component should be a mobile component or not. By promoting separation at the early development stage, application designers and programmers would have more flexibility in choosing a problem-solving technique without being restricted to the development of a function.

3.3.4 Context as Physical Environment Representation

Recall the discussion about the relationship between a mobile application and its physical environment in Section 2.3.2 that introduced the concept of *context*. In mobile applications, a context is used to represent the abstraction of an environment element perceived by a mobile entity, and can be used for

devising interaction between an entity and its physical environment. Facilitating such an interaction is important, as the performance of an application can be greatly influenced by the environment.

A context can provide information about a particular element of the physical environment. A *context value* is a piece of information that denotes a situation presently being experienced by a mobile entity that is aware of the context being represented. To illustrate the concept of context value, consider the space ship in the scenario which is aware of the location (i.e., planet) and weather contexts. At any time, the current location and weather condition experienced by the space ship can be shown by context values such as "Mars" or "meteor storm".

During a migration, a mobile entity may be experiencing a situation referred by a particular context value (e.g., being in Mars, or being in the middle of a storm). The state of experiencing such a situation is called the *context state*. A context state has a boolean value, and is always associated with a context value. It is evaluated to true if a mobile entity is experiencing the situation referred by the context state, and false otherwise. As context values change during a migration, the context state for a particular value also toggles. Therefore it can be said that a migration causes context state changes. An analogy to an ordinary program execution can be drawn here. A context value can be regarded as the content of a program counter register (e.g., the position of the register), while the associated context state refers to the corresponding state of execution indicated by the register (e.g., the state of execution when the register is at X). A migration is similar to a step of program execution; when it advances, it changes the state of program execution, much the same as a migration changes a context state.

A context can represent a single or a compound abstraction. However, as far as a mobile application concerns, a context operates as a single unit. The information provided by a context cannot be divided into smaller,

independent parts. In this case the purpose of a context is similar to that of a data structure. Both contexts and data structures act as an abstraction mechanism for entities from the problem domain so that they can be represented and manipulated in an application. The difference is that a context still maintains its physical semantics. The application that uses the context still preserves its conceptual connection to the physical environment represented by the context.

The physical semantics of a context creates a new dimension of scoping for a mobile entity that is aware of the context. A context defines a conceptual space that binds a mobile entity and its migration. The location context, for example, defines a mental image of a bounded space in which a vehicle is moving around. This kind of scoping is called *context domain*. A context domain is defined by the set of all possible values that can be generated by a context. In the space ship scenario, the domain of the location context, for instance, is all planets in the solar system (i.e., the space ship cannot go further to locations outside this domain).

A concept parallel to context domain is *context state space*. It defines context states generated from all possible context values within a particular domain. Within a context state space, it is possible to think of a context state as a *locality* for a computation. The term *contextually local to a state* is attributed to a computation that is running in that context state. Two computations are said to be contextually local to each other if they are both experiencing the same context state. This concept extends the semantics of spatial localisation to cover contexts in general. The actual meaning of localisation is defined by the context used in the expression. For example, it is now possible to have a localisation using the social context (e.g., multiple users share a common "neighbourhood" context). Contextual locality is mostly useful in controling the interaction of two mobile entities, as explained in Section 3.3.6.

Context values (and hence context states) are very useful to mobile applications because they provide the basis for a mechanism for providing environment awareness and adaptation. The idea is to make any changes in the environment become known to the application so that appropriate actions or adjustments can be performed. The mechanism is discussed in the following section.

3.3.5 Context-Based Execution Model

Recall that functionality and mobility are properties that denote the *potentials* to perform an action and to move, respectively. The possession of these properties are distinguished from their *activation*. A functional property is active when its pertaining action is being executed. Similarly, mobility is in an active state when a vehicle is moving. Activations of functionality and mobility constitute the execution of a mobile application.

As functionality is independent of mobility, it can be deduced that a function execution is also conceptually independent of a migration, and vice versa. This implies that an action can be performed while being mobile, or conversely, a migration can start in the middle of a function execution.

In Mocha, actions and migrations are triggered by factors external to a functional entity or a vehicle. This means functional entities and vehicles do not have full internal autonomy in controling their behaviour. They do not have their own initiatives in starting their actions and performing their migrations.

The activation model for functionality and mobility is not centralised either. Each mobile entity has the ability to execute its function or to move, but there is no central controling authority for activating it. Instead, an activation is controled by changes of conditions occuring within the entity's physical environment. This is where the concept of context comes in place.

Since a context is an abstraction of a physical environment element, it can be used as a "hook" for an activation. A reactive model is used for this purpose. An activation is viewed as a *reaction* upon an event generated when a mobile entity is experiencing a particular situation. A context state can be used to capture such a situation and to bind an activation to it. With this model, activation is *passively* expressed (e.g., "start action A when arriving at location L", or "move the agent to host H if the network connection speed is higher than 10 kbps").

With the environment plays an important role in initialising an activation, the execution of a mobile application is mainly controled by context handling and manipulation. This execution model provides a core awareness mechanism, and can be used to facilitate different computing requirements. For example, deterministic mobility can be achieved by predefining the context states used to trigger the execution of certain actions. A mobile agent's migration can be modeled by a location awareness scheme with a set of predefined context states representing the destination locations. Network awareness can be modeled similarly, but with a non-deterministic pattern. Finally, any adaptation mechanism is considered as a variation of an awareness scheme. The difference between the two is determined by the functionality aspect (i.e., how the application reacts to the changes of context states it is sensitive to).

3.3.6 Concurrent Mobility and Rendezvous

To solve a problem that requires mobility, one may need more than one mobile entity, each with its own migration agenda. Naturally, such a system has to exhibit some degree of concurrency for an efficient problem-solving strategy. In the perspective of a mobile application, concurrency means the ability of having multiple migrations at a particular time (i.e., launching a vehicle while others are still active). In this situation, concurrent mobility is

similar to having concurrent processes or threads in a parallel program. The difference is that concurrent mobility has the spatial semantics. Concurrency is laid on the spatial domain, and parallelism is attributed to the activities of mobile entities that cause spatial dynamics. In Mocha, only movement can cause spatial dynamics.

Concurrent mobility implies concurrent processes. When multiple vehicles are active, they bring concurrent activation of the functionalities carried by the vehicles. This concept can even be expanded to multiple levels of concurrency, for example, if an execution at a particular place involves multiple threads. However, only the top-level (i.e., mobility) concurrency is considered by Mocha.

A system with multiple vehicles in a program requires an interaction mechanism for those vehicles. The mechanism is not for the sake of the vehicles themselves, but rather for the functional entities inside them. It is quite common for a functional entity to communicate to another entity in a different vehicle (e.g., method calls between objects representing different functions). Sometimes a communication session depends on what is happening at the mobility level, and some form of synchronisation between the vehicles is required so that the communication can be performed as expected. Mocha introduces the concept of *rendezvous* for this purpose. A rendezvous is a meeting between two vehicles. It is like a handshake between two mobile entities, which has to be established first before communication between their functional entities can start.

Mocha requires that a rendezvous can take place only if both mobile entities are contextually local to each other. As explained in the previous section, they do not have to be spatially located in the same place. Instead, an interaction can take place if they are both experiencing the same context state.

A rendezvous is also subject to Mocha's reactive computation model. It cannot start by itself, instead it has to be attached to one or more contextual preconditions. The establishment of a rendezvous is triggered by a situation occuring in the environment to which both mobile entities are sensitive. This implies that they have to listen to the same context.

Not surprisingly, the most common application of rendezvous is for spatially-local communication. This kind of communication reduces the distance gap between the communicating parties, and offers cheaper costs compared to those done between remote parties. To perform it, spatial locality has to be established prior to an communication session. In other words, the communicating entities have to synchronise their migration to a common location so that their functional entities can interact locally.

The genericity of the locality constraint allows it to be applied in non-spatial contexts as well. For example, a confidential conversation session between top managers using mobile video conference application will need to set a context, for instance *top_manager*, that contains values representing each manager's login account. Conversation can only proceed if the program detects that no non-managers are joining the channel. It should be realised though that non-spatial rendezvous may involve mobile entities located at different places. Their interaction can incur expensive network operations.

If communication between functional entities is independent of what happens at the mobility level, rendezvous is not necessary. Traditional remote communication between distributed objects can be categorised into this type of communication. As it is completely decoupled from any migration activity, it is transparent at the mobility level. From a vehicle's point of view, it is seen as an ordinary functionality.

A rendezvous is initiated by a vehicle sending a request to the other party. The request/reply protocol as used in the RPC mechanism [BN84] is

used to facilitate the initialisation of a rendezvous. A vehicle requesting for a rendezvous has to wait for a reply from the other vehicle or fail the request upon a no-reply situation.

The following description analyses the protocol using the concept of logical time. From the point of view of logical timing, a rendezvous is a time synchronisation between two vehicles V_1 and V_2 so that $t_{V_1} = t_{V_2} = T_s$, where T_s is the synchronised time (i.e., the time when the rendezvous starts), while t_{V_1} and t_{V_2} are the current times of V_1 and V_2, respectively. T_s is predefined to reflect the future occurrence of the context state used to establish the rendezvous. Additionally, t_1 is the time when V_1 sends a rendezvous request to V_2, and t_2 is the time V_1 receives a reply from V_2. Figure 3.2 illustrates the time synchronisation of a rendezvous.

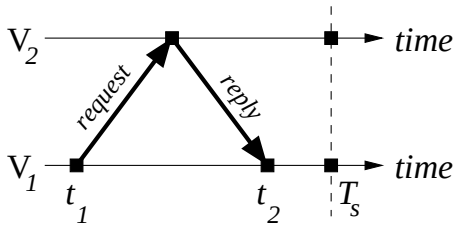

Figure 3.2: Time synchronisation in a rendezvous

A requirement for a rendezvous to be successfully initialised is that t_2 must not exceed T_s. Failure to meet this requirement might be caused by errors in either the request or reply message transmission. If the request message cannot reach V_2, or the reply cannot reach V_1 (or it arrives late), the timing requirement for t_2 cannot be fulfiled, thus failing a rendezvous plan.

To conclude this section, several concepts that are used in the Mocha framework have been introduced. Collectively these concepts show Mocha's unique paradigm for controling migrations and providing environment awareness and adaptation. It incorporates these features into a uniform context-based framework. What is required is to weave these concepts into a single model, and this will be explained in the next section.

3.4 A Context-Based Modeling Framework for Mobile Systems

This section formalises the concepts defined in the Section 3.3. The formalism of Mocha needs to have two properties to express context-awareness of a mobile entity. Firstly it has to support the state-based nature of Mocha, in which the activation of an action or a migration is based on the satisfaction of certain precondition states. Logic-based formalisms such as used in the UNITY model [CM88] and its derivatives [RMP97, WR96] fits this requirement. A system can be described by specifying its properties using a set of logic statements. The behaviour of the system can be observed by evaluating the logic statements against the system's states.

The logic-based formalism, however, cannot accommodate the structural characteristic of mobile entities in an expressive manner. A logic predicate can represent the structure of a vehicle, but it cannot capture atomic operations that can be performed upon the structure. This weakness is complemented by the spatial-based approach, which is adopted by calculus systems such as Mobile Ambients [CG98] and Agent-Places [TA98]. This approach models computation as surrounded by a hierarchical conceptual space, which has primitive operations to allow computation to move from one space to another.

Mocha's formalism combines certain aspects of the logic-based and spatial-based models. The structure of the formalism follows the logic-based framework, in the sense that a set of logic statements are used to describe a system. However, a predicate in a statement can now be composed as a structural construct representing a vehicle or a functional entity. Borrowing the idea of the spatial-based model, such a construct has *primitive actions* for movement purposes, which can be integrated into the evaluation of a structural logic predicate. Such an operation is executed automatically and

atomically during an evaluation. Using this hybrid approach, it is possible to capture both *what is going on* (i.e., shown by the expression) and *what one will have* (i.e., shown by the result of the operation).

Using the logic-based approach, system properties can be specified through UNITY's relations (see Section 3.2.1). Safety properties can be specified using the **unless**, **until**, **stable**, and **invariant** relations, while progress properties are defined using the **leads-to**, **ensures**, and **takes-to** relations. A slight modification is made to the **takes-to** relation (Statement 3.2). In Mocha a **takes-to** relation is no longer bound to a location nor to any other context, but it can transform a vehicle from one mobility state to another.

The basic assertion that describes the UNITY's computation model, i.e., $\{p\}s\{q\}$ (see Section 3.2.1), is also valid for Mocha. An execution of an action s in any state that satisfies the predicate p will result in a state that satisfites q. In Mocha, p and q refer to context states. However, given that Mocha states are closely related to the physical environment, a state that satisfies q may also be generated from outside the realm of a program's execution. This actually becomes a distinguishing characteristic of Mocha. The notion of action s that triggers a context change is expanded to cover a non-conventional form of "actions". In other words, s does not necessarily take the form of normal program statements, but it can materialise as *external forces* from the physical environment (e.g., when a context state change is caused by a non-controllable context such as network speed).

Mocha also provides a way to manipulate a set of assertions $\{p\}s\{q\}$ to define the pattern of movement of a vehicle. In the UNITY model, the default is to have no such mechanism. UNITY does not need such mechanism because it works with non-deterministic system behaviour. In such a system, the set of statements form an *open* description of the system; there is no assertion that leads to the completion of a program execution. To accommodate systems with deterministic behaviour, Mocha allows assertions to be

bound to a finite set of context states and to ensure that these states will eventually evaluate to true. This can be achieved by introducing the notion of *sequences* to complement the existing UNITY's constructor (Statement 3.1). Section 3.5.2 has more details on this issue.

3.4.1 Context Domains, Context Values, and Context States

Mocha recognises the infinite set of all context domains, $\mathbb{C}$, which is defined as follows.

$$\mathbb{C} = \{C_1, C_2, \ldots, C_j, \ldots\} \tag{3.4}$$

where $C_j,\ 1 \leq j \leq \infty$ are individual context domains.

A context domain is defined as a (possibly infinite) set of context values. The definition of both finite and infinite context domains are given below.

$$C_j^* = \{x_{j1}, x_{j2}, \ldots, x_{ji}, \ldots\}, \quad 1 \leq i \leq \infty \tag{3.5}$$

$$C_j = \{x_{j1}, x_{j2}, \ldots, x_{ji}, \ldots, x_{jm}\}, \quad 1 \leq i \leq m \tag{3.6}$$

where $1 \leq j \leq \infty$ in both cases and x_{ji} are individual context values. C_j^* is distinguished from C_j for its infinite membership. A context value x_{ji} can represent a *discrete* or *continuous* value. The difference is indicated by their mapping to *real values*. A discrete context value maps into exactly one real value, so it is safe to use a real value to represent a context value. For example, in a context domain representing a subnetwork, a context value referring to a machine is discrete, because it maps to a single real value of IP address. A continuous value, on the other hand, maps to a range of real values. A "low" context value referring to a network speed measurement falls into this category, since it maps to a certain interval of measurement results (e.g., a "low" speed means 0 to 10 kbps, for instance). Note that

Definitions 3.5 and 3.6 require that each context value must be explicitly identified. In the case of a continuous context value, a proper identifier representing the value must be assigned to denote the associated interval of real values.

A context state can be created from a context value. For each context value x_{ji}, there is a corresponding context state s_{ji}. A context state has a boolean value and its operational semantics is defined by attributing it to a vehicle V. It gets its boolean value by evaluating whether a vehicle *is experiencing* the situation referred by the corresponding x_{ji}.

$$V\langle s_{ji}\rangle \equiv V\langle s(x_{ji})\rangle = \begin{cases} \texttt{true} & \text{if } V \bullet x_{ji} \\ \texttt{false} & \text{otherwise} \end{cases} \tag{3.7}$$

Definition 3.7 shows a vehicle V is listening to a context state s_{ji}, whose value is determined by its corresponding context value x_{ji}. The $\bullet$ symbol denotes a boolean contextual operator that evaluates whether the vehicle is experiencing the specified context value. As an example, given a context value "Mars", the semantics of the corresponding context state is whether a vehicle is being in Mars or not.

3.4.2 Mobility States

The state-based mechanism of Mocha is built on top of the context state concept. Mocha introduces the concept of *mobility state*, which is derived by applying temporal attributes to a context state. Mobility state is based on logical time [Lam78]. Logical time is a one-dimensional, discrete temporal coordinate system which does not necessarily depend on actual time. The value of logical time increases at an unpredictable rate, and it never decreases. Logical time is generated by logical clocks. A logical clock is attached to a context, and activated whenever the context is operational. A Mocha system has a collection of local, uncoordinated clocks generating independent instances of logical time.

Mocha uses the notion of events to describe the semantics of a mobility state. During a time period where $V\langle s_{ji}\rangle$ is true, an event is generated. The event has a continuous spectrum, and is bordered by two time points, t_1 and t_2, that denote the state transitions for $V\langle s_{ji}\rangle$. The t_1 time point indicates a context state change that makes $V\langle s_{ji}\rangle$ become true, while the t_2 time point indicates the opposite. Both t_1 and t_2 have positive values; negative values show that they do not exist. The values of t_1 and t_2 do not necessarily depend on the current time t. A graphical illustration of time points for a particular event is shown in Figure 3.3.

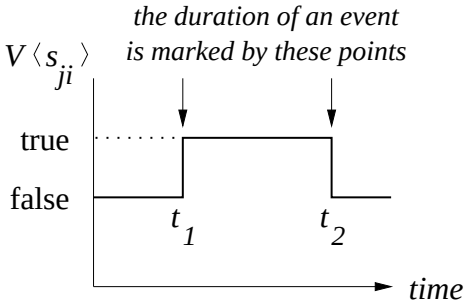

Figure 3.3: State transition for a context state and its corresponding events

Let E be an event generated by a particular situation where $V\langle s_{ji}\rangle$ is true. With respect to the time points of E, there are three possible temporal situations that a vehicle can experience: *before* E takes place, *when*[3] E is happening, and *after* E has finished. These states are called the *mobility states* of the vehicle, with respect to an event E generated by a context state s_{ji}. Their semantics is defined as follows.

$$
\begin{aligned}
V\langle before(E)\rangle &= \begin{cases} \texttt{true} & \text{if } (t < t_1), \\ \texttt{false} & \text{otherwise} \end{cases} \\
V\langle at(E)\rangle &= \begin{cases} \texttt{true} & \text{if } (t_1 \le t \le t_2), \\ \texttt{false} & \text{otherwise} \end{cases} \\
V\langle after(E)\rangle &= \begin{cases} \texttt{true} & \text{if } (t > t_2), \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.8}
$$

[3]The keyword *at* is used to denote this state.

The notation t denotes the current time of a vehicle, its value increases through the passage of actual time. The event E and its corresponding time points t_1 and t_2 are unique. For any E, the corresponding mobility states are mutually exclusive. A vehicle can only be at one state at a particular time. From Definitions 3.7 and 3.8, it can be deduced that

$$V\langle s_{ji}\rangle \equiv V\langle at(E)\rangle \tag{3.9}$$

To summarise, context states have a close relationship with mobility states. Context states are used to represent the actual dynamics of the physical environment, which is expressed in terms of context state changes. Mobility states apply temporal attributes to this representation to build the state-based mechanism of Mocha.

The causal connection between mobility states depends on the time T_0, which shows the time the logical clock for a context starts ticking (i.e., the time the context is defined and becomes operational). There are three possible situations when T_0 is compared to t_1 and t_2 of an event E. These are shown in Figure 3.4.

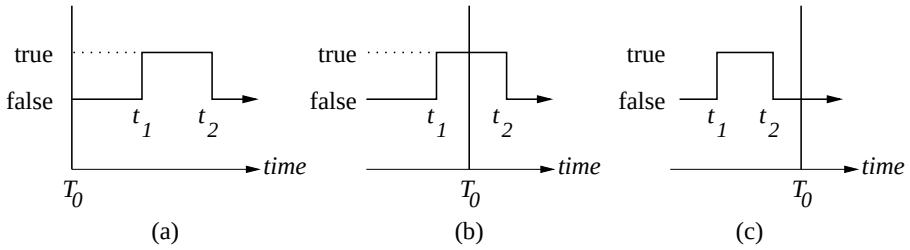

Figure 3.4: Causal connection between mobility states

In Figure 3.4-a, both t_1 and t_2 are larger than T_0. In this case, an occurrence of the *after* state implies a previous *at* state, which in turn implies a *before* state (this causality is written as $before < at < after$). In the second case (Figure 3.4-b), the event E is happening while the logical clock is started. The only valid causality is $at < after$, because the *before* state for that

particular event is not defined. Finally, the situation depicted in Figure 3.4-c will never happen, as the whole event does not exist when the context is operational (indicated by the negative values of t_1 and t_2).

To work with mobility states at the operational level, one has to be able to properly identify the event of interest. Therefore it is necessary to assign sufficient identification to an event to make it unique. Using the context value associated with the context state that triggers an event is not sufficient, because the same value can generate more than one event. For instance, the context value "Mars" can trigger multiple events; this happens when a vehicle performs multiple visits to Mars.

One possible way to achieve uniqueness is to use a sequencing mechanism to generate a unique identifier for an event occurrence. This approach can be implemented by a simple counter. Events associated with a context value has their own counter, which is incremented every time an event occurs. A unique identifier for each event can be constructed by pairing the context value and the counter, i.e., (x_{ji}, n_j). Using a system variable that holds the value of a counter, the semantics of mobility states at the operational level can be written as follows. The symbol $\stackrel{\triangle}{=}$ stands for "is abbreviated to".

$$
\begin{aligned}
V\,\langle before(x_{ji}, n_j)\rangle &\stackrel{\triangle}{=} \\
V\,\langle \diamond(x_{ji}, n_j)\rangle &= \begin{cases} \texttt{true} & \text{if } (t < t_1) \wedge (n = n_j), \\ \texttt{false} & \text{otherwise} \end{cases} \\
V\,\langle at(x_{ji}, n_j)\rangle &\stackrel{\triangle}{=} \\
V\,\langle (x_{ji}, n_j)\rangle &= \begin{cases} \texttt{true} & \text{if } (t_1 \leq t \leq t_2) \wedge (n = n_j), \\ \texttt{false} & \text{otherwise} \end{cases} \\
V\,\langle after(x_{ji}, n_j)\rangle &\stackrel{\triangle}{=} \\
V\,\langle (x_{ji}, n_j)\diamond\rangle &= \begin{cases} \texttt{true} & \text{if } (t > t_2) \wedge (n = n_j), \\ \texttt{false} & \text{otherwise} \end{cases}
\end{aligned}
\tag{3.10}
$$

The diamond symbol ($\diamond$) is a shorthand for the *before* state (if placed in front of the value pair) and *after* state (if placed after the pair). A graphical

illustration of mobility states at the operational level is given in Figure 3.5. The horizontal bars show time points where mobility states for different event occurrences yield a true value.

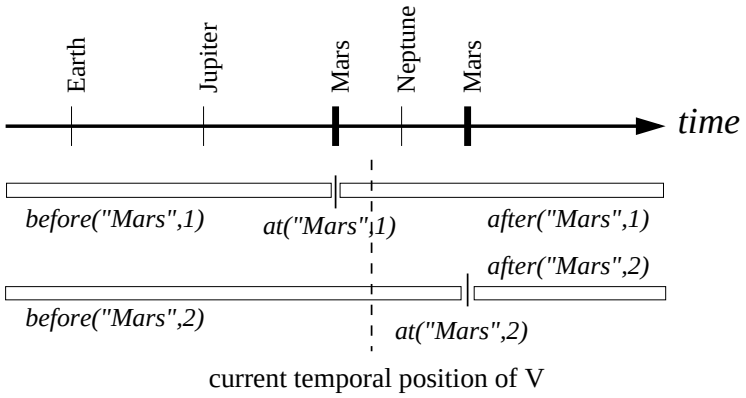

Figure 3.5: Representing mobility states

Figure 3.5 shows a vehicle V has visited Mars for the first time and is heading to Neptune before returning to Mars for the second time. In this temporal situation, the mobility states $after("Mars", 1)$ and $before("Mars", 2)$ evaluate to true, while the other mobility states yield false value.

3.4.3 Structural Representation of Vehicles

A vehicle is used to represent the mobility of an entity. It is denoted by a boolean constructor that returns true if the indicated vehicle exists or can be successfully created. The syntax of a vehicle definition, written in Extended BNF, is shown as follows.

$$
\begin{array}{lcll}
V_k & ::= & 0 & \text{null vehicle} \\
 & | & M & \text{vehicle constant (name)} \\
 & | & M[V_1, \ldots, V_m, F_1, \ldots, F_n] & \text{containment} \\
 & | & \beta.V_{k'} & \text{structuring mechanism}
\end{array}
\tag{3.11}
$$

$$\begin{array}{lcll} \beta & ::= & \textbf{in}\ V_{k''} \mid \textbf{in}\ \mathrm{P} & \text{entering capability} \\ & \mid & \textbf{out}\ V_{k''} \mid \textbf{out}\ \mathrm{P} & \text{exiting capability} \end{array}$$

The structuring mechanism allows a vehicle to enter or exit another vehicle. The mechanism is implemented by two internal features called *capabilities*, borrowed from the Mobile Ambient calculus [CG98]. A capability is different from a functionality, although both imply the notion of action. A capability is instantaneously activated when its expression is evaluated (i.e., no explicit activation is required). Its execution is performed atomically. Evaluation of a capability expression yields a true result if the operation is successful, otherwise a false result is produced. The latter is due either to the non-existence of the parent, or the refusal of the parent for the child to enter to or exit from it.

Mocha's containment is different from that of Mobile Ambient as it does not contain running processes. Instead it handles *dormant* functions and other containment structures (i.e., other vehicles). However, the **in** and **out** capabilities have the entering and exiting semantics similar to those of Mobile Ambient. The semantics of a successful operation is given as follows.

$$V_{k'}[V_2] \wedge V_k[V_1] \wedge\ \textbf{in}\ V_{k'}.V_k \quad \rightarrow \quad V_k[V_1, V_{k'}[V_2]] \tag{3.12}$$

$$V_k[\ \textbf{out}\ V_{k'}[V_2].V_k, V_1] \quad \rightarrow \quad V_k[V_1] \wedge V_{k'}[V_2] \tag{3.13}$$

Statement 3.12 states that in a system with vehicles $V_{k'}$ and V_k with their respective children V_2 and V_1, the **in** $V_{k'}.V_k$ capability places $V_{k'}$ and its content as a child of V_k. The opposite action is described by Statement 3.13 where the **out** $V_{k'}[V_2].V_k$ capability moves $V_{k'}[V_2]$ out of V_k so they become siblings. For functions, the entering and exiting mechanisms are described by the following reductions.

$$V_k[\ ] \wedge \mathrm{P} \wedge\ \textbf{in}\ \mathrm{P}.V_k \quad \rightarrow \quad V_k[\mathrm{P}] \tag{3.14}$$

$$V_k[\ \textbf{out}\ \mathrm{P}.V_k] \quad \rightarrow \quad V_k[\ ] \wedge \mathrm{P} \tag{3.15}$$

3.4.4 Context-Awareness

Mocha captures the relationship between a vehicle and a set of contexts into an association between the vehicle and a set of mobility states derived from the contexts. Given M a set of mobility state expressions m_i (refer to Definition 3.10), the relationship can be shown as follows.

$$V_k \langle M \rangle \equiv V_k \langle m_1, \ldots, m_n \rangle \tag{3.16}$$

where m_i is either $\diamond(x_{ji}, n_j)$, (x_{ji}, n_j), or $(x_{ji}, n_j)\diamond$.

An association of a vehicle and a set of mobility states represents a mobile entity that listens to one or more contexts and is interested in one or more events from the contexts' domain. A context-aware vehicle expression returns true if the structural property of the vehicle can be created and its association with all its mobility states is also satisfied. That is,

$$V \langle m_i \rangle \rightarrow V \wedge \langle m_i \rangle \tag{3.17}$$

Moreover,

$$\neg V \langle m_i \rangle \rightarrow \neg V \vee \neg \langle m_i \rangle \tag{3.18}$$

and this should be distinguished from

$$V \langle \neg m_i \rangle \rightarrow V \vee \langle \neg m_i \rangle \tag{3.19}$$

The expression $\neg\langle m_i \rangle$ in Statement 3.18 indicates that a vehicle V is not associated with the context state referred by m_i (i.e., the expression returns a false association). It is used to refer a vehicle's loss of association with the context referred by m_i. In this case, whether m_i refers to a `before`, `at`, or `after` does not really matter. To illustrate, the expression $Ship \langle (\text{"}Mars\text{"}, *)\diamond \rangle$ yields true if the space ship does exist and it has left Mars (the '*' symbol indicate a *don't care* value, which matches to any value). The expression $\neg Ship \langle (\text{"}Mars\text{"}, *)\diamond \rangle$ means that either the space ship does not exist, or it

has nothing to do with destination planets. Finally, $Ship\langle\neg(\text{"}Mars\text{"}, *)\diamond\rangle$ is true if the space ship *has not left* Mars (i.e., either has not arrived at Mars, or is currently visiting it).

A context-awareness association only applies to top-level vehicles. Children vehicles are not affected by the scheme (i.e., they are "frozen" inside a parent vehicle). So in an expression

$$V_1[V_2\ \langle m_2\rangle\ ]\ \langle m_1\rangle\ , \tag{3.20}$$

the awareness mechanism only applies to V_1 (i.e., only the association with m_1 is in effect). In this case, it is safe to omit the non-significant part of an expression, so Statement 3.20 could be written as follows.

$$V_1[V_2]\ \langle m_1\rangle \tag{3.21}$$

3.4.5 Functionality Representation

Functionality represents the ability of performing an action. Like a vehicle, a functional entity is represented as a constructor that yields a boolean value. It returns a true value if the indicated functionality is present. The syntax of a functionality definition is shown as follows.

$$\begin{array}{lcll} F_i & ::= & 0 & \text{null functionality} \\ & | & \text{P} & \text{action constant (function name)} \\ & | & \alpha_s & \text{request communication} \\ & | & \alpha_r & \text{reply communication} \end{array} \tag{3.22}$$

The α functionality indicates the ability of performing communication. It is a property of a vehicle. Therefore, only vehicles can communicate with each other. Vehicle communication is only used for rendezvous purposes. The request mode transmits request messages, while the reply mode has the semantics of acknowledging and approving the request.

Some form of communications may also be performed during the execution of an action, for example, in method invocation between objects. This kind of communication is, however, outside the scope of Mocha, because such a communication is performed intrinsically as part of an action.

3.4.6 Functionality and Mobility Activation

The execution of a function is represented by a sequence of *activation* and *deactivation*. An activation denotes the state of being active (executed), while a deactivation refers to the state of being inactive after an activation has completed.

Activation and deactivation of an action are represented by the $\overline{F_i}$ and $\underline{F_i}$ predicates, respectively. The expressions evaluate to true if the activation and deactivation can be successfully carried out. They have a **leads-to** relation (i.e., $\overline{F_i} \mapsto \underline{F_i}$). The relationship implies that an activation will always lead to a deactivation, either gracefully (normal deactivation) or abruptly (abnormal termination). Moreover, a deactivation expression can be implicitly hidden behind an activation expression (i.e., given an expression $\mathrm{P} \rightarrow \overline{\mathrm{P}}$, it is not necessary to state $\overline{\mathrm{P}} \mapsto \underline{\mathrm{P}}$). However, certain situations may require explicit deactivation expression, for instance, when the execution of Q cannot start until P completes. In this case, $\neg\overline{\mathrm{Q}}$ **unless** $\underline{\mathrm{P}}$.

Contrary to the functionality aspect, activation of mobility can be expressed implicitly or explicitly. An implicit migration is expressed through its effects on the vehicle's contexts. Given two expressions, $V\,\langle(x_{j1}, n_{j1})\rangle$ and $V\,\langle(x_{j2}, n_{j2})\rangle$ with $x_{j1} \neq x_{j2}$ and $x_{j1}, x_{j2} \in C_j$, if during the passage of time, both $V\,\langle(x_{j1}, n_{j1})\rangle$ and $V\,\langle(x_{j2}, n_{j2})\rangle$ have ever evaluated to true, then one or more context state changes have occured. Using the assumption of indirect location representation (see Section 3.1), the context state changes can be used to express a migration.

An explicit migration expression is represented by an action predicate that takes two arguments indicating the initial and final states. It is used when it is necessary to put the emphasis on context state changes caused by the migration. The following example illustrates a migration from one location to another.

$$Go(i,j) \textbf{ takes } V\,\langle m_i \rangle \textbf{ to } V\,\langle m_j \rangle \tag{3.23}$$

The predicates $V\langle m_i \rangle$ and $V\langle m_j \rangle$ show the mobility states prior and after the action execution, respectively. Once an action predicate has been defined, it can be activated by making it evaluate to true.

3.4.7 Context-Based Mobile Systems

A Mocha system can be modeled by a collection of logic statements. The reactivity of a system's computation is naturally accommodated by the underlying execution model that performs evaluation of the statements fairly often over an infinite time period. Functionality and mobility activation are bound to the context-awareness scheme, as shown below.

$$\begin{aligned} V[\mathrm{P}]\,\langle m_i \rangle &\rightarrow V[\overline{\mathrm{P}}]\,\langle m_i \rangle \\ V[\mathrm{P}]\,\langle m_i \rangle &\rightarrow Go(i,j) \end{aligned} \tag{3.24}$$

The expressions state that, upon a true value of the left part, the corresponding action and migration is executed in the contextual scope set by m_i. For example, the basic feature of a mobile agent that searches information servers for some data can be specified by the following statements.

$$\begin{aligned} &Agent[\mathrm{FindInfo}]\,\langle dest, * \rangle \rightarrow Agent[\overline{\mathrm{FindInfo}}]\,\langle dest, * \rangle \\ &Go(source, dest) \textbf{ takes } Agent[\mathrm{FindInfo}]\,\langle source, * \rangle \textbf{ to} \\ &\quad Agent[\mathrm{FindInfo}]\,\langle dest, * \rangle \end{aligned}$$

$$Agent[\underline{\text{FindInfo}}]\ \langle dest, * \rangle\ \rightarrow (source = dest) \wedge (dest = *) \wedge$$
$$Go(source, dest)$$

Recall that the context-aware mechanism only works on top-level vehicle constructs (see Section 3.4.4). Action and migration activation consequently follow this rule. The consequence is shown by the following statement, expanded from Statement 3.20.

$$V_1[V_2[\mathrm{P}]\ \langle m_i \rangle\ , \mathrm{Q}]\ \langle m_i \rangle\ \rightarrow V_1[V_2[\mathrm{P}]\ \langle m_i \rangle\ , \overline{\mathrm{Q}}]\ \langle m_i \rangle \tag{3.25}$$

3.4.8 Rendezvous

A rendezvous is initialised by a pair of handshaking messages between the meeting vehicles. The activation of message transmissions also follows the context-based mechanism. The vehicles have to listen to the same context, but they do not have to use the same set of mobility states.

$$V_1[\mathrm{P}, \overline{\alpha_s}]\ \langle m_i \rangle$$
$$V_2[\mathrm{Q}, \overline{\alpha_r}]\ \langle m_j \rangle$$

The activation of handshaking is terminated either by a match or a mismatch between them. A match occurs if the communication link can be established (i.e., a request matches a reply). A mismatch can be caused by transmission errors, a refusal to meet (from a sender's perspective) or an unacknowledged reply (from a receiver's perspective). A deactivation due to a mismatch is denoted by the symbol $\underline{\alpha_s}_e$ or $\underline{\alpha_r}_e$, respectively. Stated formally,

$$V_1[\mathrm{P}, \overline{\alpha_s}]\ \langle m_i \rangle\ \mapsto (V_1[\mathrm{P}, \overline{\alpha_s}]\ \langle * \rangle\ \wedge V_2[\mathrm{Q}, \overline{\alpha_r}]\ \langle * \rangle\ ) \vee V_1[\mathrm{P}, \underline{\alpha_s}_e]\ \langle * \rangle \tag{3.26}$$
$$V_2[\mathrm{Q}, \overline{\alpha_r}]\ \langle m_j \rangle\ \mapsto (V_1[\mathrm{P}, \overline{\alpha_s}]\ \langle * \rangle\ \wedge V_2[\mathrm{Q}, \overline{\alpha_r}]\ \langle * \rangle\ ) \vee V_2[\mathrm{Q}, \underline{\alpha_r}_e]\ \langle * \rangle \tag{3.27}$$

The '*' symbol denotes that the mobility state indicating a handshake termination is not important.

Once a match occurs, a rendezvous can be established. The **sync** predicate is used to indicate a rendezvous.

$$V_1[\mathrm{P}, \overline{\alpha_{ms}}]\ \langle m_i \rangle\ \wedge V_2[\mathrm{Q}, \overline{\alpha_{mr}}]\ \langle m_j \rangle\ \rightarrow\ \textbf{sync}\ (V_1[\mathrm{P}], V_2[\mathrm{Q}])\ \langle m_k \rangle \quad (3.28)$$

$$\neg\ \textbf{sync}\ (V_1[\mathrm{P}], V_2[\mathrm{Q}])\ \langle m_k \rangle\ \textbf{unless}\ V_1[\mathrm{P}, \overline{\alpha_{ms}}]\ \langle * \rangle\ \wedge V_2[\mathrm{Q}, \overline{\alpha_{mr}}]\ \langle * \rangle \quad (3.29)$$

Statement 3.28 states that a matching communication, indicated by the same communication functionality identifier of V_1 and V_2 (i.e., α_m), will lead to a rendezvous. The **sync** predicate shows that the vehicles are experiencing the same mobility state (i.e., m_k). Here m_k could be the same as or different from either m_i or m_j. Statement 3.29 is a complement for Statement 3.28. It states that a matching communication as well as the same mobility state are prerequisites for a rendezvous between two vehicles. There is a subtle difference between the two rules which leads to different applications of the rules. The former looks at a rendezvous as a consequence of a state evaluation. It is used to deduce the occurrence of a rendezvous upon the establishment of a communication link (indicate by the α messages) and a true value resulted from the associated mobility state evaluation. The latter states that a rendezvous will not take place unless its preconditions are not satisfied. This rule is normally used to verify whether a rendezvous can occur or not.

Once a rendezvous is established, communication between vehicles can be started. As this communication is completely under the scope of an action execution, it is represented by activating the corresponding functionalities from both vehicles.

$$\textbf{sync}\ (V_1[\mathrm{P}], V_2[\mathrm{Q}])\ \langle m_k \rangle\ \rightarrow V_1[\overline{\mathrm{P}}]\ \langle m_k \rangle\ \wedge V_2[\overline{\mathrm{Q}}]\ \langle m_k \rangle \quad (3.30)$$

A communication between two functions from different vehicles can be completely made transparent at the mobility level if it is independent

of the mobility of the vehicles (e.g., conventional remote communications). Otherwise, if the communication is associated with an activity at the mobility level, it has to be encapsulated using the rendezvous mechanism. For example, in the space ship scenario, the wandering robot cannot enter the vehicle to bring back more sample result until a transmission of current analysis result to a remote computer is completed. This situation is described by the following statement. It states that a rendezvous is maintained until the transmission completes.

$$\begin{aligned} &\textbf{sync}\ (\mathit{Ship}[\mathrm{T1}], \mathit{0}[\mathrm{T2}])\ \langle (i,*)\rangle\ \wedge \neg\ \textbf{in}\ \mathit{Robot}[\ldots].\mathit{Ship}\ \langle (i,*)\rangle \\ &\quad \textbf{until}\ \mathit{Ship}[\underline{\mathrm{T1}}]\ \langle (i,*)\rangle\ \wedge\ \mathit{0}[\underline{\mathrm{T2}}]\ \langle (i,*)\rangle \end{aligned} \tag{3.31}$$

The data transmission functionality is denoted by T1 and T2. The remote computer is represented by *0*, indicating its immobility. The expression $\neg$ **in** *Robot*[...].*Ship* $\langle (i,*)\rangle$ denotes a failure in the entering operation due to the block of the space ship.

As explained in Section 3.3.6, a rendezvous is constrained by the context of the participating vehicles because the underlying communication depends on the context state when the communication is taking place. Consequently, contextual locality between the vehicles must be maintained during the rendezvous. Therefore a *strict* policy has to be enforced. The policy states that once a rendezvous is started in a particular mobility state, it will remain in that state until the underlying functions finish the communication. A context state that changes the mobility state is not allowed. The strict policy can only be applied if the vehicles have complete control over the dynamics of their contexts.

A strict rendezvous can be specified by maintaining the synchronised state until functions complete the communication.

$$\textbf{sync}\ (V_1[\mathrm{P}], V_2[\mathrm{Q}])\ \langle m_i\rangle\ \ \textbf{until}\ \ V_1[\underline{\mathrm{P}}]\ \langle m_i\rangle\ \wedge V_2[\underline{\mathrm{Q}}]\ \langle m_i\rangle \tag{3.32}$$

The rule states that a rendezvous should be preserved until the functions finish with their interaction (denoted by deactivating the actions).

If the above rule is not present, then the rendezvous is a *loose* one. A loose rendezvous does not put any constraint on context state change during a rendezvous. A context state change is allowed to take place when two vehicles are meeting. This happens if the dynamics of the context cannot be controlled by the mobile application. For example, in a network performance test application, two mobile agents can be sent to different nodes to measure the quality of the connection between the two nodes. The agents are programmed to perform the test based on different categories of connection quality. In this situation, rendezvous is made on the network connection quality context, which is completely outside the control of the agents.

3.5 Modeling Mobile Applications

This section illustrates how Mocha can be used to model mobile applications. Three examples are provided. The first example illustrates how the basic properties of space ship scenario can be modeled. The other two examples show how Mocha deals with deterministic and non-deterministic mobility.

3.5.1 Modeling the Space Ship Scenario

The following illustration shows how Mocha can be used to specify the space ship scenario described in Section 3.3.1. It shows how to model a pattern of movement, context-based function execution, and context awareness and adaptation behaviour of the system described in the scenario.

The laboratory tests, autopilot, and sample collection functionalities are abbreviated to LT, AP, and SC, respectively. **INIT** is a special predicate that indicates the initial system configuration. All variables are free, their

values can change between statement evaluations. The initial configuration of the system can be expressed as follows.

$$\begin{aligned}\textbf{INIT} \;\rightarrow\; & \mathit{Ship}[\,]\,\langle(\text{"}\mathit{Earth}\text{"},1)\rangle \;\wedge\; \textbf{in}\;\mathrm{LT}.\mathit{Ship} \;\wedge\; \textbf{in}\;\mathrm{AP}.\mathit{Ship} \;\wedge \\ & \textbf{in}\;(\mathit{Robot}[\,]\,\langle(\text{"}\mathit{Earth}\text{"},1)\rangle \;\wedge\; \textbf{in}\;\mathrm{SC}.\mathit{Robot}).\mathit{Ship} \qquad (3.33)\end{aligned}$$

The next rule states that once the ship is being in a planet, it will eventually move to another. This eventuality is expressed using the **ensures** relation. Variables i and j denote destination locations and they range over all planets that can be visited by the space ship. The $\exists$ symbol denotes that the ship only considers traveling only to some planets.

$$\begin{aligned}&\langle\; \exists i,j \;::\; && (3.34)\\ &\quad \mathit{Ship}[\mathrm{LT},\mathrm{AP},\mathit{Robot}[\mathrm{SC}]]\,\langle(i,*)\rangle \;\textbf{ensures}\\ &\qquad \mathit{Ship}[\mathrm{LT},\mathrm{AP},\mathit{Robot}[\mathrm{SC}]]\,\langle(j,*)\rangle \;\wedge (i \neq j)\\ &\;\rangle\end{aligned}$$

The above expression shows a progress property of the space ship. It defines an allowable state transition from a given state. However, it does not specify any instance of migration that produces the state transition. The action that actually moves the space ship is specified using the *Go* predicate, similar to that shown by Statement 3.23.

$$\begin{aligned}&\mathit{Go}(i,j)\;\textbf{takes}\;\mathit{Ship}[\mathrm{LT},\mathrm{AP},\mathit{Robot}[\mathrm{SC}]]\,\langle(i,*)\rangle \;\wedge (i \neq j)\;\textbf{to} \qquad (3.35)\\ &\quad \mathit{Ship}[\mathrm{LT},\mathrm{AP},\mathit{Robot}[\mathrm{SC}]]\,\langle(j,*)\rangle\end{aligned}$$

The *Go(i,j)* action predicate returns true when the space ship is traveling between two planets indicated by i and j.

When the space ship arrives at a planet, it unloads the robot, and the astronaut can start working on the lab tests. These are denoted by the **out** capability and the activation of the functionality, respectively.

$$Ship[\mathrm{LT}, \mathrm{AP}, Robot[\mathrm{SC}]]\ \langle(i,*)\rangle\ \rightarrow \tag{3.36}$$
$$Ship[\mathrm{LT}, \mathrm{AP},\ \textbf{out}\ Robot[\mathrm{SC}].Ship]\ \langle(i,*)\rangle\ \wedge Ship[\overline{\mathrm{LT}}, \mathrm{AP}]\ \langle(i,*)\rangle$$

When the **out** capability in Statement 3.36 is in effect, the robot leaves the space ship and starts its own mobility, creating concurrent mobility. The robot's journey can be specified in a similar manner to that of the space ship. For example, the specification can be derived from Statements. 3.34, 3.35, and 3.36 with appropriate modifications (e.g., on the specifications of the robot vehicle and its destinations).

The space ship is not allowed to leave a planet until the lab tests in that session are completed and the robot has entered the ship again. This restriction is specified as a safety property using the **unless** relation.

$$\neg Go(i,j)\ \textbf{unless}\ Ship[\underline{\mathrm{LT}}, \mathrm{AP}]\ \langle(i,*)\rangle\ \wedge\ \textbf{in}\ \ Robot[SC].Ship\ \langle(i,*)\rangle \tag{3.37}$$

Finally, a travel between two planets is specified by a rule that triggers an action denoted by the *Go* predicate.

$$Ship[\underline{\mathrm{LT}}, \mathrm{AP}, Robot[\mathrm{SC}]]\ \langle(i,*)\rangle\ \rightarrow Go(i,j) \tag{3.38}$$

Note that the space ship's journey does not use an explicit itinerary as no specific route is defined. A stronger sequential sense can be modeled using the sequential property of a constructor. This is discussed in the next section.

3.5.2 Deterministic Mobility

Deterministic mobility is characterised by the sequential property of a pattern of movement. In a mobile agent application, for example, deterministic pattern is shown by the agent's itinerary, expressed in terms of a series of migration procedures to the specified destinations.

The state-based nature of Mocha is not naturally suitable for representing deterministic systems, because it does not have a mechanism to control the evaluation of its statements to follow a deterministic pattern. This situation is inherited from UNITY; each statement in a UNITY program is selected arbitrarily but with weak fairness (i.e., each statement is selected infinitely often in an infinite computation) [RMP97].

Modelling deterministic mobility in Mocha therefore requires a mechanism that allows sequencing of context state changes. This is achieved by extending the semantics of the UNITY's constructor (Definition 3.1) with a sequential property. A Mocha constructor can accept a sequence of values, indicated by a pair of square brackets, in its range constraint. The order of these values is significant. The operational semantics is defined by two new operators, called *successor* and *predecessor*, which are denoted by the $\succ$ and $\prec$ symbols, respectively. Given a constructor

$$\langle\ \forall i, j\ : i \in [x_1, \ldots, x_n]; j \succ i\ :: expr \rangle, \tag{3.39}$$

j will always be assigned the successor value of i, in the order given by the bracketed expression. Their values are then used to construct the expression *expr*.

The following example uses a mobile agent to illustrate modeling of a deterministic system. For example, the agent has a function which has to be executed at the destination locations A, B, and C. The mobile agent's movement and the activation of its function can be represented as follows.

$$\begin{aligned}
&\langle\ \forall i, j\ : i \in [A, B, C]; j \succ i\ :: \\
&\quad Go(i, j)\ \textbf{takes}\ Agent[\text{Func}]\ \langle (i, 1) \rangle\ \ \textbf{to}\ Agent[\text{Func}]\ \langle (j, 1) \rangle \\
&\quad Agent[\text{Func}]\ \langle (i, 1) \rangle\ \rightarrow Agent[\overline{\text{Func}}]\ \langle (i, 1) \rangle \\
&\quad Agent[\underline{\text{Func}}]\ \langle (i, 1) \rangle\ \wedge (\ i \neq C) \rightarrow Go(i, j)\ ) \\
&\ \rangle
\end{aligned} \tag{3.40}$$

The constructor contains three statement templates which will be used to create actual statement expressions according to the specified quantification of variables i and j. The order of the sequence of A, B, and C is significant. The expression $j \succ i$ yields the next sequence value of the one denoted by i, if exists, which is then assigned to j during statement construction. If such a value does not exist (i.e., i holds the last element in the sequence), j will be empty and this will fail any logical expression that uses it. When the constructor is evaluated, actual statement expressions are created from the templates and they will reflect the agent's sequential behaviour.

A variation of the above situation is when the destination of the agent is not predefined but can be computed instead (e.g., as a result of the `Func` function). In this case the system is still deterministic, but its sequential behaviour cannot be predetermined anymore. This is the case of the example described in Section 3.5.1. In this situation, there is no need to use a constructor to describe such a system, as shown in the following statement.

$$Agent[\text{Func}]\ \langle (j,1) \rangle\ \rightarrow Agent[\overline{\text{Func}}]\ \langle (j,1) \rangle \tag{3.41}$$

The variable j is a free variable. It gets its value from the result of `Func`. Context state change (i.e., migration to another location) may happen between evaluations of the statement, but it does not matter because the agent is not interested in it.

3.5.3 Non-Deterministic Mobility

Non-deterministic mobility occurs when the pattern of movement of a mobile entity cannot be predefined or computed at all. Movements are performed randomly, and it is not possible to determine the next move using currently available information. Consider, for example, modeling the mobility of a salesman. His travel pattern cannot be defined or planned as it depends on the job order on a particular day.

In Mocha, non-deterministic mobility is translated to the inability to determine the pattern of context state changes, either at compile-time (pre-defined pattern or at run-time. However, Mocha allows a non-deterministic system to be specified without dealing with the uncertainty. The state-based model can capture the definable behaviour of the system and leave the uncertain part out of the specification. In the case of the salesman, it is only necessary to specify the context states in which the application is interested in. For example, the salesman sells his goods and always makes his way to a cafe for lunch. This can be expressed by the following rules. Note that he only comes for lunch on the first visit to the cafe (indicated by the value of the counter in the mobility state expression).

$$SMan[\text{eat}, \text{sell}]\,\langle(\text{"}Cafe\text{"}, 1)\rangle \;\rightarrow SMan[\overline{\text{eat}}, \text{sell}]\,\langle(\text{"}Cafe\text{"}, 1)\rangle \quad (3.42)$$

$$SMan[\text{eat}, \text{sell}]\,\langle(x, *)\rangle \;\wedge (x \neq \text{"}Cafe\text{"}) \rightarrow \quad (3.43)$$

$$SMan[\text{eat}, \overline{\text{sell}}]\,\langle(x, *)\rangle$$

The travel pattern of the salesman is not important in this situation, and the state-based model leaves it out of the the specification.

Non-determinism can also be raised by applications that use non-controllable contexts. The dynamics of the contexts is completely outside the control of the application, and context state changes can occur at any time with no specific pattern. To illustrate how Mocha models this situaion, an application that is sensitive to network speed is used. It listens to the network transfer rate context and is aware of changes between the "high" and "low" states. It has a functionality P1 which is executed if the connection quality is good, otherwise it automatically switches to execute P2, and vice-versa. Both P1 and P2 represent the same action (e.g., video conferencing) but with different modes of operation (e.g., colour vs black-and-white). Initially,

$$\textbf{INIT} \;\rightarrow App[\,]\,\langle(*, *)\rangle \;\wedge\; \textbf{in}\ \text{P1}.App \;\wedge\; \textbf{in}\ \text{P2}.App \quad (3.44)$$

Soon after becoming operational, the context-awareness mechanism is activated. If the current context value is "low", then P2 is executed, otherwise P1 is activated. This is expressed as follows.

$$App[\mathrm{P1}, \mathrm{P2}]\ \langle (low, *) \rangle\ \rightarrow App[\mathrm{P1}, \overline{\mathrm{P2}}]\ \langle (low, *) \rangle \tag{3.45}$$

$$App[\mathrm{P1}, \mathrm{P2}]\ \langle (high, *) \rangle\ \rightarrow App[\overline{\mathrm{P1}}, \mathrm{P2}]\ \langle (high, *) \rangle \tag{3.46}$$

The automatic switching behaviour is specified in the following rules. Note how a transition of execution from P1 to P2 and vice versa is modeled. When P1 or P2 is active, it remains so until there is a change of network speed. If it happens, the current functionality is deactivated and replaced by the activation of the other one.

$$App[\overline{\mathrm{P1}}, \mathrm{P2}]\ \langle (high, *) \rangle\ \textbf{until}\ App[\overline{\mathrm{P1}}, \mathrm{P2}]\ \langle (low, *) \rangle \tag{3.47}$$
$$App[\overline{\mathrm{P1}}, \mathrm{P2}]\ \langle (low, *) \rangle\ \rightarrow App[\underline{\mathrm{P1}}, \overline{\mathrm{P2}}]\ \langle (low, *) \rangle$$

$$App[\mathrm{P1}, \overline{\mathrm{P2}}]\ \langle (low, *) \rangle\ \textbf{until}\ App[\mathrm{P1}, \overline{\mathrm{P2}}]\ \langle (high, *) \rangle \tag{3.48}$$
$$App[\mathrm{P1}, \overline{\mathrm{P2}}]\ \langle (high, *) \rangle\ \rightarrow App[\overline{\mathrm{P1}}, \underline{\mathrm{P2}}]\ \langle (high, *) \rangle$$

The system does not even have any progress property, because its behaviour is completely reactive. A context state can be non-deterministically changed by anything that affects the network performance. Also note that the mobility of the application is not expressed using the notion of locations, but instead is completely hidden behind the dynamics within the network speed context.

3.6 Discussion

Section 2.6 discussed the requirements that need to be satisfied by a model so that it can provide comprehensive and uniform support for specifying and controling the mobility aspect of a mobile application. The requirements are reiterated and shown as follows.

- the model should exhibit a generic perspective that views mobility as an independent abstraction,
- the model should capture the abstraction of physical environment aspects in a computation, and
- the model should separate the mobility and functionality aspects of a mobile application.

The rest of this section analyses how Mocha accomplishes the specified requirements and discusses their effects on mobile application development.

3.6.1 Genericity

One of the goal of this research is to avoid distinction of supporting mobility on the basis of the principal of migration. Genericity is required to provide uniform support for different requirements posed by user-oriented and code-oriented mobile applications. A generic perspective of mobility is implemented by capturing the most essential feature of any mobile entity. Functionality is selected as this representative aspect, since any entity with mobility potential has the ability to perform an action. Mobility abstraction is then built on top of the functionality representation.

The approach taken by Mocha reverses the relationship between mobility and functionality commonly found in traditional development tools. Mobility is no longer an intrinsic attribute of an entity, but instead becomes a subject that controls the activation of functionalities. In this perspective, the mobility abstraction occupies the higher level, while the details of an action is hidden behind the functionality abstraction, lying on a lower level and outside the scope of Mocha.

The top-down approach of controling mobility hides principal-specific differences of requirements that have to be handled by mobile applications.

It allows mobile application development to follow a natural way (i.e., to start with an abstract mobility model), rather than to awkwardly start it with programming-specific requirements. Furthermore, the approach allows implementation-specific decisions to be delayed. Given a set of functions that need to be run remotely, the *how* aspect of the remote execution can be specified in a later stage. For instance, design considerations made in the design stage could be used to decide whether remote task execution has to be implemented by human migration, code migration, or even traditional techniques such as remote evaluation [SG90].

3.6.2 Context-Based Model

The close relationship between a mobile entity and its physical environment is also captured by Mocha. Environment awareness is a basic property of any mobile entity. This also applies even to mobile code; its mobility is based on the concept of location, and location is one of numerous environment elements that can be abstracted by the concept of context. Context is a powerful abstraction, since it allows a movement to be expressed in terms of its effects on the environment. It is now possible to use environment elements other than locations as the basis of reasoning in a computation. Again, genericity brings in a positive effect in this situation. It enhances current development techniques (especially for code mobility) to cover environment awareness in a simple way. For example, a mobile agent can be easily equipped with some form of intelligence to sense its environment.

Mocha treats adaptation to environment as a logical consequence of an awareness scheme. From the model level, there is no difference between them, since they are distinguished by their reaction towards an environmental change. Awareness leads to an acknowledgement, while adaptation triggers a self-regulating action. From Mocha's point of view, this is a functional issue, and it is handled at the functional level. Awareness and adaptation is handled

using the same model. This approach is simpler than other approaches that requires additional layers to model the scheme [BG98, WB98, NPS95].

Context is the concept that unifies a mobile entity, its environment, and the activation of its functionalities. A context-based mechanism is created to establish an association between action execution and the dynamics within the environment. The mechanism uses a reactive model where action execution is triggered by the satisfaction of a set of precondition states. The reactive model is selected to facilitate movements that have non-deterministic patterns. The same execution framework is also used for the environment awareness and adaptation schemes.

The reactive model may seem to be inappropriate for certain types of mobile applications which exhibits a strong algorithmic sense, such as mobile agent applications. The behaviour of an agent is often expressed as a series of algorithmic steps that move the agent to a location and execute its tasks in that location. Mocha handles the situation by using a sequencing mechanism that operates on patterns of movement. With this arrangement, the dynamics within the environment is forced to behave deterministically by following a sequential pattern, producing the required effect.

3.6.3 Separation of Mobility from Functionality

The importance of separating mobility from the functionality aspect has been discussed in Chapter 2. Mocha is aware of this issue, and support is given through the distinction of mobility and functionality abstractions. They are treated separately by creating different abstraction levels for them and drawing a clear boundary between the levels.

Separating mobility and functionality issues into different levels of abstraction has a significant advantage. It offers more flexibility in further development stages by allowing implementation of mobility specification to be

conducted completely independent of implementation of functionality. This would not be possible if separation is performed in the same abstraction level, as the implementation must be bound to a common platform, which is usually defined by the programming language. For example, implementation of aspects in AspectJ [LK98] is heavily influenced by the Java language, which is used to develop the functional aspect. The idea of separating mobility from functionality has been tested to propose a visual programming framework for mobile applications [SNS00]. The framework is based on a metaphor similar to the space ship scenario described in this chapter. It aims to reduce unnecessary coupling between mobility and functionality programming, as well as to ease programming through visual manipulation.

To conclude this chapter, this section has revisited the requirements for a modeling framework and discussed how Mocha satisfies them. Mocha successfully fulfils its goals as a modeling framework, but it is not sufficiently useful as a specification tool yet. The latter goal is significantly important in a development process, because it describes a model as a set of requirement specifications that are going to be implemented in the later stage. A good specification tool creates good models that can minimise the abstraction gap between the modeling stage and the stages beyond it. The next chapter discusses the effort to develop a specification tool based on the framework presented in this chapter.

Chapter 4

Mocha: A Specification Language for Mobility Control

4.1 Introduction

In a software development project, there are certain requirements on functions, performance, interface, and constraints that have to be met by the program. They are specified using *models* that depict information, processes, system behaviour, and other software characteristic in graphical or textual symbols. Given a set of requirements, the focus of a development effort is to implement a solution to meet the requirements. In the implementation stage, high-level, user-oriented abstractions defined in a model have to be refined and transformed into program code for machine consumption.

In the traditional approach, modeling of mobility as part of application development has not received much attention. Mobility is not regarded as an abstraction with significant importance to the application being developed. This leads developers to represent mobility-related requirements using existing modeling framework, such as the flow model (Data Flow Diagram [Pre97] and Control Flow Diagram [HP87]), the Class-Responsibility-Collaborator (CRC) model [WBWW90], or the Unified Modeling Language (UML) [RJB99].

The problem with the existing modeling tools is that they are not specifically designed to model mobility. They cannot naturally capture the unique properties of mobile systems. Their modeling abstractions cannot be used to comprehensively model a mobile system, preventing developers from working with mobility using a holistic perspective. They only offer *partial* modeling where a model describes only a particular aspect of the system. When this situation is brought to the implementation stage, programming is unnecessarily confined to a narrower scope. It is normally centered around supporting a dominant type of principal of migration, and programming support is provided following this direction as well. Problems arise when more complex mobility-related requirements are defined. The programming framework cannot handle this situation well due to its partial approach.

Mocha can help solve this problem through its ability to capture the essential properties of mobile systems. A description about the functions, constraints, and behaviour of a mobile system can be specified in a holistic manner. No sacrifice to favour certain mobility aspects has to be made, and programmers can have a more comprehensive description about the application being developed. From here, a more well-informed implementation strategy can be developed.

A Mocha model has the role to stand as a bridging layer between the modeling and implementation stages. It has to be represented in such way that the model's abstractions can fit into an abstraction refinement process. They have to be easily understood and translated into more detailed representations. On the other side, a model representation has to keep all the semantics of Mocha, i.e., no loss of semantics is allowed.

In its intermediary position, a Mocha model defines an implementation framework in which the development of mobility-related components are separated from that of functional components. The idea of decoupling the development domains is due to the orthogonality between these components.

The orthogonal relationship can be explained through a software reuse perspective. In software reuse, there is an assumption that a component is created to be reusable. The assumption exists (e.g., in the form of an interface specification) even before the actual piece of code is designed and programmed. In the Mocha modeling paradigm, however, prior assumption as practiced in the reuse framework does not have to exist. A mobility control can be specified without having to know about the functionality that uses it (e.g., an itinerary does not have to be related to a particular task executed at the visited places). Conversely, a function can be developed without having to consider about its potential of being mobile. It does not matter whether the presence of one aspect will affect the development of the other or not.

What is required by Mocha to perform its role is a representation tool. Mocha uses a *specification language* to represent a model. The linguistic approach is selected for several reasons. Firstly, it is relatively easy to design a language with sufficient expresiveness to model Mocha's paradigm (i.e., to express Mocha's abstractions and their relationships). Secondly, the linguistic approach is also capable of modeling the orthogonality of mobility and functionality, while at the same time provides a uniform composition mechanism for abstractions from both aspects. Finally, a linguistic model can naturally minimise the abstraction gap between itself and the programming language used to implement the model. This is necessary to facilitate a smooth refinement process in the development stages beyond modeling.

The research described in this chapter focuses on the design of the specification language, which is also called **Mocha**. It starts with the specification framework and some design considerations. The syntax and semantics of the language is then presented, followed by the semantic description of Mocha's linguistic constructs. Following this is the section that explains the usage of the language through some illustrations. Finally some issues towards the implementation are presented to conclude this chapter.

4.2 Mocha Specification Framework

As described in Chapter 3, Mocha views mobility and functionality as two distinct abstractions. The presence of these abstractions requires the specification language to facilitate two distinct representation models: the state-based model for specifying mobility control and the algorithmic model for function specification. It has been known that they are not compatible to each other. The state-based model favours a declarative approach where system behaviour is expressed in terms of declarative statements. In this approach, there is no significant ordering that reflects execution flow, which on the other hand becomes the emphasis of the algorithmic model.

Mocha handles the multi-paradigmatic situation by assigning different abstraction levels for mobility and functionality representation. It concentrates on the specification of mobility control, which occupies the higher level. The lower level is outside the scope of Mocha. It is the place where the programming details of all function abstractions are specified.

Mocha distinguishes two different specification components: passive components that represent mobile entities carrying some functional abstractions, and control components that manipulate passive components. Therefore mobility control specification involves two different activities: *definition* of linguistic constructs representing passive components, and *manipulation* of these constructs by control statements.

The relationship between these activities are comparable to procedural programming paradigm, in which data becomes the subject of process manipulation. However, the relationship between passive components and their controls is not intrusive. When implemented in an executable program, mobility control does not modify the internal value of any passive component. Instead, it is limited to controling function executions and relocation of passive components.

Functional abstractions are specified using the algorihmic approach. These abstractions may be specified at the mobility level, however, they are not first-class. Their algorithmic representation is restricted so that it does not interfere with the control specification. There is a limit on how deep an algorithmic abstraction can be defined, and this limit is set by Mocha's linguistic constructs to prevent one from writing too detailed specification. Alternatively, actions can be specified outside Mocha's scope and a function reference can be made using the function's signature (i.e., its name and list of arguments).

Passive components are manipulated using a state-based model. The execution model follows a reactive model. Each action or movement is bound to one or more precondition states, which abstract the concept of mobility state. An activation is started when the preconditions hold (i.e., when the mobility states become true).

State evaluation is controled by control statements. A Mocha specification can have more than one control statement, each of them has its own control abstraction. Mocha does not have a specific evaluation order. Statements are evaluated in an arbitrary order, and multiple statements trigger concurrent evaluation.

Whenever possible, algorithmic processes in mobility control should be disguised in a more abstract form. This can be done using a sequencing abstraction. For instance, a series of visits can use a sequence of *values* that denote the locations to be visited, instead of using a series of *go* actions. This way the algorithmic steps are implicitly hidden behind the property of the sequence. Using this approach, a travel itinerary (or a pattern of movement in general) can be written in a more concise way. More importantly, potential conflicts between algorithmic and declarative specifications at the mobility level can be minimised.

To conclude this section, the process of specifying mobility control in Mocha can be summarised as follows.

1. *Definition of functional and mobility components.*
   Any action must be specified (using limited high-level abstraction) or referred (its definition is specified elsewhere). The vehicles that carry these functions must also be defined.

2. *Definition of contexts.*
   Context domains, as well as their context values, that will be used in a context-based execution mechanism must be defined.

3. *Definition of context statements.*
   This is the place where vehicle and function objects are composed with context objects. A context statement provides state-based hooks for action execution. This is achieved by setting up execution rules represented by mobility state expressions.

4.3 Language Design Decision

Mocha is designed for mobility control specification. The orthogonality between mobility and functionality implies that Mocha's design has to clearly support the separation of mobility from functionality. Some decisions on certain design aspects have to be made to satisfy the above requirement. They are presented as follows.

1. *Specification paradigm.*
   Mocha can be thought as an *abstract object-based* language. The object-based paradigm is adopted to minimise the abstraction gap between Mocha and existing development tools. Considering that many modeling frameworks and programming languages use the object-based (including object-oriented) paradigm, developers should not encounter

much difficulty in using Mocha in a development environment based on this paradigm.

Mocha does not work with actual instances of objects, but instead operates on *abstract* objects that exist only at the specification level. However, Mocha objects need not be interpreted using the traditional concept of object in object-oriented programming languages. They have the semantics of a generic data abstraction, so they can be realised by different data models used by programming languages. This approach allows Mocha to be designed like an actual programming language while maintaining its abstract nature.

2. *Simple language architecture.*

   When mobility is isolated from functionality, it is possible to specify mobility control using simple and concise expressions. This requires a simple language architecture with limited number of constructs with high-level semantics. Mocha satisfies this requirement by using only two basic constructs that represent passive system elements (e.g., vehicles and functions) and controling elements. It is sufficient to express any mobility control requirement using *objects* and *statements*.

   Although Mocha looks like a programming language, it is not. Therefore it is necessary to avoid using implementation-related concepts in the language (e.g., the use of variables to hold values). This is to maintain the abstractness of the language. In a similar spirit, operations that lead to functional data manipulation must not be provided.

   Mocha recognises separate abstraction levels for mobility and functionality specification. However, Mocha components operate only at the mobility level. Mocha does not facilitate algorithmic specification unless it is directly related with mobility. Detailed function specification is made at the functionality level, which is outside the scope of Mocha.

3. *High-level and abstract semantics.*

   To compensate the limited number of constructs, it is necessary to assign high-level and abstract semantics to them. At the same time, this may implicitly confine the application of the constructs to mobility-related matters. While it seems to be restrictive, one should be reminded that the constructs are not designed for general specification purposes.

4.4 Mocha Language Specification

This section describes Mocha specification language in details. First the structure of Mocha programs and the object model are presented. This is followed by the description of the syntax and semantics of Mocha's linguistic constructs. A syntax is given using the Extended BNF format. The following conventions of symbols are used.

- A non-terminal syntactic element is shown by enclosing it in a pair of angle brackets (⟨ ⟩).
- A terminal node is indicated in boldface (if it represents a keyword) or enclosed in single quotes (if it represents a syntactic symbol).
- A range of values is shown by the doble dots (..) symbol.
- The [ ] pair symbols indicate optional syntactic elements.
- The | symbol represents alternatives.
- The ( )* symbols indicate zero or more occurrence of the enclosed elements.
- The ( )+ pair indicate one or more occurrence of the enclosed elements.

It should be noted that the presentation of the syntax is for illustrative purpose. Priority is given to the clarity of the presentation so that it can clearly describe the structure of a Mocha program and how it is composed from its components. Consequently, some details may be omitted, and the grammar may not be in its most efficient form to be used in creating a parser for the language. The actual grammar used in the construction of a syntax analyser for Mocha is given in Appendix A of the thesis.

4.4.1 Program Structure

A Mocha program consists of two parts: definition of objects representing context-based system elements and context statements that control the behaviour of the system elements. Each part is realised by a set of linguistic constructs that can be composed to form a complete specification program. The following syntax shows the structure of a Mocha program.

$$\text{program} \quad ::= \quad (\ \langle \text{object\_def} \rangle \ \mid \ \langle \text{context\_stmt} \rangle\ )* \tag{4.1}$$

The $\langle$object_def$\rangle$ and $\langle$context_stmt$\rangle$ are *top-level* constructs that represent object definitions and context statements, respectively. A top-level construct is the unit of abstraction in a Mocha specification program. Object and context statement definitions can appear in any order. There is no special construct that marks the *execution entry point* as commonly found in programming languages.

Objects represent entities that can be manipulated by context statements. The entities represent elements related to both mobility and functionality aspects. For example, a data item, a function that processes it, and their mobility property are all represented by objects. Different semantics are distinguished by assigning different types to objects.

4.4.2 Object Model

With respect to its object-based paradigm, Mocha uses a model similar to that of the Python language [vR00]. The concept of object is used to encapsulate the abstraction of all entities possibly represented in Mocha. Objects have *types*, *identities*, and *values*. Types and identities are invariant. The abstract nature of Mocha objects does not prevent them from having these attributes. In fact, the concepts of types, identities, and values form part of the specification. They require an implementation language to have a type system that can support them.

Object Definition

All object definitions, except functions, are defined using a uniform format, shown as follows.

$$\begin{aligned} \text{obj\_def} \quad &::= \quad \langle \text{object\_id} \rangle \ \text{' : '} \ \langle \text{type\_def} \rangle \\ &\qquad\quad [\ \text{' := '} \ \langle \text{value\_expr} \rangle\ ]\ \text{';'} \\ \text{type\_def} \quad &::= \quad [\ \langle \text{type\_id} \rangle \ \text{' = '}\ ]\ \langle \text{typedef} \rangle \end{aligned} \tag{4.2}$$

An object identifier indicated by the ⟨object_id⟩ is unique within a program's name space. The ⟨typedef⟩ represents the notation of a particular type definition. A value, denoted by ⟨value_expr⟩ , can optionally be assigned as the object's initial value. If such a value expression is not given, the object is uninitialised. The ⟨type_id⟩ tag denotes a type identifier of a type indicated by the ⟨typedef⟩ tag. A type could be predefined or user-defined. The type identifier can be used to refer a user-defined type in an object definition.

Object and Its Value

An object can obtain its value through a mechanism called *object assignment*. Object assignment is an operation that transfer the value of a source object to a recipient object. Object assignment is achieved through an assignment statement.

An indirect operation occurs in an assignment statement that involves a non-trivial expression (e.g., an arithmetic operation). This is shown by the following expression.

⟨object_id⟩ := ⟨expression⟩

In the above operation, the expression is evaluated, then its value is transparently assigned to an intermediate, anonymous object. Finally a transfer of value is performed on between the intermediate object and the recipient.

For objects of certain types, an assignment can be specified in terms of *ranges*. A range is an enumeration of objects. Only cardinal (countable) types (integer, character, boolean, and IP address) can have ranges. A range is denoted by a tilde symbol (∼) between the lower and upper bound values of the range.

Unlike actual programming languages, Mocha does not have the concept of variables. Programming languages normally use a variable to contain the reference to an object representation in memory. Because there is no concept of execution environment in Mocha, there is no need for such a container.

Types

Mocha categorises objects into two broad types: passive objects and function objects. Passive objects represent data and other components that become the object of manipulation, while functions represent executable actions.

Types for data objects are further divided into *compound* and *primitive* types. Primitive types represent a single type only. They include numbers, strings, boolean, a type to represent IP address, and a type to indicate voidness. There is only one compound type, which defines an aggregate that contains heterogenous types. Function objects are instantiated from the function type.

Types also define allowable operations on objects. Mocha insists type checking to be performed in any operations. The type of an object in an expression must meet the type requirement of the operation it is involved in. Whether type checking is performed statically or dynamically is an implementation issue and the decision is left to an implementation language.

One could observe that type declaration is dependent on (i.e., as part of) object definition. It only exists if there is a need to create an object of that type. An implication of this situation is that a user-defined type only exists if there is an explicit need for an object of that type within the scope of a program.

The reason to make type definition hide behind object definition is that a Mocha specification mainly concerns with state-based manipulation of objects with a high level abstraction. It is important to highlight object representation and manipulation as the main theme in the specification. Representation of any object attribute should not obscure the overall program structure, so that it becomes difficult to grasp the high-level abstraction of objects. In this situation, providing a rigorous typing mechanism in the mobility level is less important than program expresiveness and conciseness. Typing, on the other hand, is important in the implementation of the specification. It is required to ensure type safety which is imposed by many programming languages. The approach taken by Mocha tries to pursue both goals. By having a single object-type definition, Mocha will only work with two essential constructs: objects and state-based control mechanism. At

the same time, the "hidden" type information that accompanies an object definition can be used to impose type checking in the implementation level.

4.4.3 Type Specification

This section describes each type known in Mocha. The syntax and semantics of a type is explained in each subsection. Syntax presentation is grouped by types.

Integer

The integer type is designated by the keyword **int**. An integer object represents a signed integer number. The range of possible values for integers is not defined by Mocha, but instead by an implementation language. The syntax of an integer object definition is as follows.

$$
\begin{array}{rcl}
\text{integer\_def} & ::= & \langle \text{object\_id} \rangle \ \text{':'} \ \langle \text{int\_typedef} \rangle \\
 & & [\ \text{':='}\ \langle \text{int\_expr} \rangle\ ]\ \text{';'} \\
\text{int\_typedef} & ::= & (\ [\ \langle \text{type\_id} \rangle\ \text{'='}\ ]\ \textbf{int}\ )\ |\ \langle \text{type\_id} \rangle \\
\text{int\_expr} & ::= & \langle \text{int\_arg} \rangle\ (\ \langle \text{i\_op} \rangle\ \langle \text{int\_arg} \rangle\ )* \\
\text{int\_arg} & ::= & \langle \text{integer\_literal} \rangle\ |\ \langle \text{object\_id} \rangle\ | \\
 & & \langle \text{function\_call\_expr} \rangle \\
\text{integer\_literal} & ::= & [\ \text{'+'}\ |\ \text{'-'}\ ]\ \langle \text{digit} \rangle\ (\ \langle \text{digit} \rangle\ )* \\
\text{digit} & ::= & \textbf{1..9} \\
\text{i\_op} & ::= & \text{'+'}\ |\ \text{'-'}\ |\ \text{'*'}\ |\ \text{'/'}\ |\ \text{'\%'}
\end{array}
\tag{4.3}
$$

The ⟨function_call_expr⟩ tag represents a function call expression and its syntax is given in Definition 4.14. The following examples show some instantiation of integer objects.

```
anInt : intType = int := 1;
anotherInt : int := 10;
cloneInt : int := anInt;
num : int := anotherInt * aFunction() - 100;
```

Float

The float type is used to represent floating point numbers. It has double precision, and the range is defined by the implementation language. It is denoted by the keyword **float**. The syntax of a float definition is given in the following.

$$
\begin{array}{rcl}
\text{float\_def} & ::= & \langle \text{object\_id} \rangle \text{ ':' } \langle \text{float\_typedef} \rangle \\
 & & [\text{ ':=' } \langle \text{float\_expr} \rangle] \text{ ';'} \\
\text{float\_typedef} & ::= & ([\langle \text{type\_id} \rangle \text{ '='}] \textbf{ float }) \mid \langle \text{type\_id} \rangle \\
\text{float\_expr} & ::= & \langle \text{float\_arg} \rangle \ \langle \text{f\_op} \rangle (\langle \text{float\_arg} \rangle \mid \langle \text{int\_arg} \rangle) * \\
\text{float\_arg} & ::= & \langle \text{float\_literal} \rangle \mid \langle \text{integer\_literal} \rangle \mid \\
 & & \langle \text{object\_id} \rangle \mid \langle \text{function\_call\_expr} \rangle \\
\text{f\_op} & ::= & \text{'+'} \mid \text{'-'} \mid \text{'*'} \mid \text{'/'} \\
\text{float\_literal} & ::= & [\text{ '+'} \mid \text{'-'}] (\langle \text{digit} \rangle) * \text{ '.'} (\langle \text{digit} \rangle) + \\
 & & [\langle \text{exponent} \rangle] \\
 & \mid & [\text{ '+'} \mid \text{'-'}] \text{ '.'} (\langle \text{digit} \rangle) + [\langle \text{exponent} \rangle] \\
 & \mid & [\text{ '+'} \mid \text{'-'}] (\langle \text{digit} \rangle) + [\langle \text{exponent} \rangle] \\
\text{exponent} & ::= & \textbf{E} \mid \textbf{e} [\text{ '+'} \mid \text{'-'}] (\langle \text{digit} \rangle) +
\end{array}
\tag{4.4}
$$

The ⟨function_call_expr⟩ tag represents a function call expression and its syntax is given in Definition 4.14. The definition of ⟨digit⟩ is given in Definition 4.3. Type coercion occurs if an operation involves both integer and float arguments. In this situation, the integer arguments are converted to floats. Some examples of floating-point number definition follow.

```
aFloat : float := 10.0;
anotherFloat : float := aFloat / 4;
f : float := anInt + 10.35;
f1 : float := a_float_function();
```

String

A string type represents a series of alphanumeric characters. Characters represent at least 8-bit bytes, and normally bytes 0-127 represent the ASCII encoding. However the interpretation of the character bytes is left to an implementation language. The type is denoted by the keyword **string**. The syntax of a string type definition is shown below.

$$
\begin{array}{rcl}
\text{string\_def} & ::= & \langle \text{object\_id} \rangle \ \text{':'} \ \langle \text{string\_typedef} \rangle \\
 & & [\ \text{':='} \ \langle \text{string\_expr} \rangle\ ]\ \text{';'} \\
\text{string\_typedef} & ::= & (\ [\ \langle \text{type\_id} \rangle \ \text{'='}\ ]\ \textbf{string}\ ) \mid \langle \text{type\_id} \rangle \\
\text{string\_expr} & ::= & \langle \text{string\_arg} \rangle \ \ (\ \text{'+'} \ \langle \text{string\_arg} \rangle\ )* \\
\text{string\_arg} & ::= & \langle \text{string\_literal} \rangle \mid \langle \text{object\_id} \rangle \mid \\
 & & \langle \text{function\_call\_expr} \rangle \\
\text{string\_literal} & ::= & \textbf{"} \ \langle \text{string} \rangle \ \textbf{"} \\
\text{string} & ::= & (\ \langle \text{alpha\_chars} \rangle\ )+ \\
\text{alpha\_chars} & ::= & \textbf{0..9} \mid \textbf{A..Z} \mid \textbf{a..z}
\end{array}
\tag{4.5}
$$

The syntax shows that a string can be constructed by a concatenation operation, indicated by the '+' symbol. The ⟨function_call_expr⟩ tag represents a function call expression and its syntax is given in Definition 4.14.

There is no separate type to denote a single character. A character is represented by a string with one element. The following code shows some string definition examples.

```
aString : string := "a";
anotherString : string := aString;
str1 : string := "a" + "string";
```

Boolean

The boolean type is used to create boolean objects, which have the values of true and false. The type is designated by the keyword **boolean**. The syntax for boolean object definition is as follows.

$$\begin{aligned} \text{boolean\_def} &::= \langle\text{object\_id}\rangle \text{ ' : ' } \langle\text{bool\_typedef}\rangle \\ &\quad [\text{ ' := ' } \langle\text{bool\_expr}\rangle \text{] ';'} \\ \text{bool\_typedef} &::= (\text{ [} \langle\text{type\_id}\rangle \text{ ' = '] } \textbf{boolean} \text{) } \mid \langle\text{type\_id}\rangle \\ \text{bool\_expr} &::= \langle\text{logical\_expr}\rangle \mid \langle\text{comp\_expr}\rangle \end{aligned} \tag{4.6}$$

The ⟨logical_expr⟩ and ⟨comp_expr⟩ node represent logical and comparison expressions which yield boolean values, and they are described below.

$$\begin{aligned} \text{logical\_expr} &::= \langle\text{bool\_arg}\rangle \quad (\langle\text{log\_op}\rangle \ \langle\text{bool\_arg}\rangle)* \\ \text{bool\_arg} &::= [\text{ '!' }] (\langle\text{boolean\_literal}\rangle \mid \langle\text{object\_id}\rangle \mid \\ &\quad \langle\text{function\_call\_expr}\rangle) \\ \text{boolean\_literal} &::= \textbf{true} \mid \textbf{false} \\ \text{log\_op} &::= \&\& \quad | \quad || \end{aligned}$$

The !, &&, and || symbols represent the NOT, AND, and OR operators, respectively. The syntax for ⟨function_call_expr⟩ is given in Definition 4.14.

Comparison operations work with integers, floats, or strings, and return a boolean value. A comparison is performed on the values of the compared objects.

$$\begin{aligned} \text{comp\_expr} &::= \langle\text{comp\_arg}\rangle \quad \langle\text{comp\_op}\rangle \quad \langle\text{comp\_arg}\rangle \\ \text{comp\_arg} &::= \langle\text{int\_arg}\rangle \mid \langle\text{float\_arg}\rangle \mid \\ &\quad \langle\text{string\_arg}\rangle \\ \text{comp\_op} &::= > \quad | \quad >= \quad | \quad == \quad | \quad <= \quad | \quad < \quad | \quad != \end{aligned} \tag{4.7}$$

The ⟨int_arg⟩ , ⟨float_arg⟩ , and ⟨string_arg⟩ tags are defined in Definitions 4.3, 4.4, and 4.5, respectively. The semantics of each comparison operator depends on the type of the arguments. Numeric arguments are compared by their numeric value, while string arguments are compared by their internal representation (e.g., ASCII value).

The following examples shows how boolean objects can be created using different kinds of boolean expressions.

```
aBool : boolean := true;
anotherBool : boolean := false;
b1 : boolean := min < max;
lost : boolean := !found;
```

IP Address

An object of IP address type represents an Internet address. It has a value which can be expressed either in the 4-tuple number format or the string format. The range of the values is defined by the Internet addressing system. The IP address type is denoted by the **ipaddr** keyword.

$$
\begin{array}{rcl}
\text{ipaddr\_def} & ::= & \langle\text{object\_id}\rangle \text{ ' : ' } \langle\text{ipaddr\_typedef}\rangle \\
 & & [\text{ ' := ' } \langle\text{ipaddr\_expr}\rangle \text{] ';'} \\
\text{ipaddr\_typedef} & ::= & (\text{ [} \langle\text{type\_id}\rangle \text{ ' = '] } \textbf{ipaddr} \text{) } | \langle\text{type\_id}\rangle \\
\text{ipaddr\_expr} & ::= & \langle\text{ipaddr\_literal}\rangle \mid \langle\text{object\_id}\rangle \mid \\
 & & \langle\text{function\_call\_expr}\rangle \\
\text{ipaddr\_literal} & ::= & \langle\text{ip\_num}\rangle \mid \langle\text{ip\_string}\rangle \\
\text{ip\_num} & ::= & \langle\text{ipn}\rangle \text{ '.' } \langle\text{ipn}\rangle \text{ '.' } \langle\text{ipn}\rangle \text{ '.' } \langle\text{ipn}\rangle \\
\text{ipn} & ::= & \textbf{0..255} \\
\text{ip\_string} & ::= & \langle\text{string}\rangle \text{ ('.' } \langle\text{string}\rangle \text{) } *
\end{array}
\tag{4.8}
$$

The ⟨function_call_expr⟩ tag is a function call expression whose syntax is given in Definition 4.14. The ⟨string⟩ tag is defined in Definition 4.5.

An IP address object represents an abstraction that belongs to the environment of a mobile entity. The reason to represent the IP-based location context as a language primitive is that it is so commonly used in mobile computing. Including it as part of Mocha language specification allows easy and straightforward representation of locations in many mobile applications.

One can mix the use of the 4-tuple number or the string format of an IP address, but two equivalent addresses (i.e., they refer to the same location according to the Domain Name System) refer an equal value of an IP address object. In other words, the *actual* value of an IP address object can be expressed using different *representations*. Detection of the validity of a value representation (i.e., whether it refers to the same machine) is left to the implementation program. As shown in the following example, if `192.168.10.5` and `myhost` point to the same machine, then both objects are said to have the same value.

```
anIP : ipaddr := 192.168.10.5;
anotherIP : ipaddr := myhost;
```

The unique aspect of the IP address type is that it can be used to define the scope of a computation. The type represents a unit of physical abstraction that denotes a location, and Mocha uses this type to define its spatial execution semantics (i.e., to bind a computation to a location). It means that given an IP address object that is used as a context element, computation is locally bound to that location.

The ability to express the binding of a computation to a physical location changes the perspective adopted by many mobile applications with code mobility. Currently most development tools for this type of applications assume that control for code migration is done by a program that launches the code from the machine where the program is executed. Since the program becomes the controller for any remote execution, it is not possible to totally transfer the execution to the migrating computation. For example, any

exception in a remote location will mostly be regarded as the program's exception, and exception handling will be performed centrally by the program. The program's central role prevents programmers to have a computation running independently on different locations, completely detached from the program and constructing its own (spatial) execution environment. The IP address type overcomes this problem by offering a mechanism for locality required for spatial binding through lexical constructs. In other words, it helps break the connection between the controling program and the migrating computation.

Void

The type void is used to represent voidness or an instance of non-existence. There is only one single instance of this object, which can only have one value indicating the voidness. The object can be accessed through its name, `void`. The object is predefined, and no other object definition of this type is possible.

Dummy Objects

A dummy object is one that has no name, therefore it cannot be referred. It is denoted by the undescore (_) symbol, and can have any type. A dummy object is used in situations where only one aspect of an object is required, i.e., either its presence, type, or value. For example, in a function definition, parameter specification often requires a handy way to refer to a type (especially for complex types). Similarly, a user-defined type may be required to accept the return value. Dummy objects allows consistent application of Mocha's object model throughout its linguistic constructs.

The type and value of a dummy object solely depends on the context of the linguistic expression where the object is used. If it is used in an object

definition, its type and value follow what is specified in the definition. If it is used in an expression, its type and value follow the required type and the result of the operation accordingly. If the context does not provide sufficient information, a dummy object acquires no particular type and value.

Set

A set represents an unordered collection of unique objects with the same type. The type of a set is determined by the type of its element objects. A set object is defined using the keyword **set**.

$$
\begin{array}{rcl}
\text{set\_def} & ::= & \langle \text{object\_id} \rangle \ \text{' : '} \ \langle \text{set\_typedef} \rangle \\
 & & [\ \text{' := '} \ \langle \text{set\_expr} \rangle\ ]\ \text{';'} \\
\text{set\_typedef} & ::= & (\ [\ \langle \text{type\_id} \rangle \ \text{' = '}\ ]\ \textbf{set} \\
 & & (\ \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{boolean} \mid \textbf{ipaddr} \mid \\
 & & \langle \text{seq\_typedef} \rangle \mid \langle \text{set\_typedef} \rangle \mid \langle \text{bag\_typedef} \rangle\ )) \\
 & \mid & \langle \text{type\_id} \rangle \\
\text{set\_expr} & ::= & \langle \text{set\_arg} \rangle \ (\ \langle \text{set\_op} \rangle \ \langle \text{set\_arg} \rangle\ )* \\
\text{set\_arg} & ::= & \langle \text{set\_literal} \rangle \mid \langle \text{object\_id} \rangle \mid \langle \text{function\_call\_expr} \rangle \\
\text{set\_literal} & ::= & \text{'\{' '\}'} \mid \\
 & & \text{'\{'} \ \langle \text{set\_elm} \rangle \ (\ \text{','} \ \langle \text{set\_elm} \rangle\ )* \ \text{'\}'} \\
\text{set\_elm} & ::= & \langle \text{int\_expr} \rangle \mid \langle \text{float\_expr} \rangle \mid \langle \text{string\_expr} \rangle \mid \\
 & & \langle \text{boolean\_expr} \rangle \mid \langle \text{ipaddr\_expr} \rangle \mid \langle \text{bag\_expr} \rangle \mid \\
 & & \langle \text{seq\_expr} \rangle \mid \langle \text{set\_expr} \rangle \mid \langle \text{range\_expr} \rangle
\end{array}
\tag{4.9}
$$

The ⟨range_expr⟩ represents a range expression whose definition is given in Definition 4.11. A set constructed from a range expression is expanded to include all elements within the range. The syntax of ⟨seq_typedef⟩ and ⟨bag_typedef⟩ are given in Definitions 4.10 and 4.12, respectively, which also define the syntax of ⟨seq_expr⟩ and ⟨bag_expr⟩ .

Symbol	Operations
+	Set union
|	Set intersection
-	Set difference
?	Set membership test
==	Set equivalence test
!=	Set inequivalence test

Table 4.1: Set operations

Table 4.1 describes the set operators. The first three operations return a set, while the remaining operations return a boolean value.

The value of a set is defined by all its members. They can be implicitly specified by stating their type, or explicitly given by enumerating them. A null set has no member. Set members must be unique, and their order of specification does not matter. The following code show some set definitions.

```
setA : set int := {1,2,3,4,5};
setB : set float := { float };          # infinite set of float
setAB : setint := setA + setB;          # set union
subnet := {192.168.10.10~50};           # constructed from a range
mynet := {[10,20,30,40,50]};            # constructed from a sequence
```

Sequence

A sequence object represents a finite ordered set of objects with the same type. The order is shown by the position of the object in the sequence. A sequence is denoted by the keyword **seq**. The syntax is shown as follows.

$$
\begin{aligned}
\text{sequence\_def} \quad ::= \quad & \langle \text{object\_id} \rangle \text{ ' : ' } \langle \text{seq\_typedef} \rangle \\
& [\text{ ' := ' } \langle \text{seq\_expr} \rangle] \text{ ';'} \\
\text{seq\_typedef} \quad ::= \quad & (\ [\langle \text{type\_id} \rangle \text{ ' = ' }]\ \textbf{seq} \\
& (\ \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{boolean} \mid \textbf{ipaddr} \mid \\
& \langle \text{seq\_typedef} \rangle \mid \langle \text{set\_typedef} \rangle \mid \langle \text{bag\_typedef} \rangle)) \\
\mid \quad & \langle \text{type\_id} \rangle
\end{aligned}
\tag{4.10}
$$

$$
\begin{array}{rcl}
\text{seq\_expr} & ::= & \langle\text{seq\_literal}\rangle \mid \langle\text{object\_id}\rangle \mid \langle\text{function\_call\_expr}\rangle \\
\text{seq\_literal} & ::= & \text{'['} \ \text{']'} \mid \\
 & & \text{'['} \ \langle\text{seq\_elm}\rangle \ (\ \text{','} \ \langle\text{seq\_elm}\rangle\ )* \ \ \text{']'} \\
\text{seq\_elm} & ::= & \langle\text{int\_expr}\rangle \mid \langle\text{float\_expr}\rangle \mid \langle\text{string\_expr}\rangle \mid \\
 & & \langle\text{boolean\_expr}\rangle \mid \langle\text{ipaddr\_expr}\rangle \mid \langle\text{bag\_expr}\rangle \mid \\
 & & \langle\text{seq\_expr}\rangle \mid \langle\text{set\_expr}\rangle \mid \langle\text{range\_expr}\rangle
\end{array}
$$

The syntax of ⟨set_typedef⟩ and ⟨bag_typedef⟩ are given in Definitions 4.9 and 4.12, respectively. The definitions also define the syntax of ⟨set_expr⟩ and ⟨bag_expr⟩. Value assignment can be done by explicitly enumerating the values, separated by commas and enclosed by a pair of square brackets ([ ]).

A sequence can also be constructed from a *range*. The syntax of range expression is as follows.

$$
\begin{array}{rcll}
\text{range\_expr} & ::= & \langle\text{int\_range}\rangle \mid \langle\text{bool\_range}\rangle \mid & (4.11) \\
 & & \langle\text{ipaddr\_range}\rangle & \\
\text{int\_range} & ::= & \langle\text{int\_arg}\rangle \ \text{'}\sim\text{'} \ \langle\text{int\_arg}\rangle & \\
\text{bool\_range} & ::= & \langle\text{boolean\_arg}\rangle \ \text{'}\sim\text{'} \ \langle\text{boolean\_arg}\rangle & \\
\text{ipaddr\_range} & ::= & \langle\text{ipn}\rangle \ \text{'}\sim\text{'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle & \\
 & | & \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'}\sim\text{'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle & \\
 & | & \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'}\sim\text{'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle & \\
 & | & \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'.'} \ \langle\text{ipn}\rangle \ \text{'}\sim\text{'} \ \langle\text{ipn}\rangle &
\end{array}
$$

The order of a range becomes the order of the sequence. This mode of sequence construction, however, can only be applied to objects with a cardinal type.

The following examples show different ways to construct a sequence. Note that when a sequence is constructed from a set, its order is arbitrarily defined.

```
seqF : seq float := [1.0,1.5,2.0,2.5];
machines := [192.168.10.11~50];
matrix : seq seq int :=
   [[1,2],[3,4],[5,6]];    # multi-dimensional sequence
seqSet : seq string := [{"Mars","Earth","Venus"}];
```

Mocha does not impose a specification on the size of a sequence. The size can be static (i.e., fixed) or dynamic (i.e., can be shrinked or enlarged).

Bag

A bag is designed to wrap multiple objects into a single unit of abstraction. Containment can recurse to any level of depth, and there is no restriction on the number and type of objects that can be loaded into a bag. The bag type is designated by the **bag** keyword.

$$
\begin{array}{rcl}
\text{bag\_def} & ::= & \langle\text{object\_id}\rangle\ \text{':'}\ \langle\text{bag\_typedef}\rangle \\
 & & [\ \text{':='}\ \langle\text{bag\_expr}\rangle]\ \text{';'} \\
\text{bag\_typedef} & ::= & (\ [\langle\text{type\_id}\rangle\ \text{'='}\ ]\ \textbf{bag}\ \langle\text{bag\_body}\rangle\ ) \\
 & | & \langle\text{type\_id}\rangle \\
\text{bag\_body} & ::= & \text{'\{'}\ (\\
 & & [\ \textbf{dynamic}\ ]\ \langle\text{object\_id}\rangle\ \text{':'}\ \langle\text{typedef}\rangle\ \text{';'} \\
 & &)*\ \text{'\}'} \\
\text{typedef} & ::= & \langle\text{int\_typedef}\rangle \mid \langle\text{float\_typedef}\rangle \mid \\
 & & \langle\text{string\_typedef}\rangle \mid \langle\text{boolean\_typedef}\rangle \mid \\
 & & \langle\text{ipaddr\_typedef}\rangle \mid \langle\text{set\_typedef}\rangle \mid \\
 & & \langle\text{seq\_typedef}\rangle \mid \langle\text{bag\_typedef}\rangle \mid \langle\text{function\_def}\rangle \\
\text{bag\_expr} & ::= & \text{'('}\ \text{')'}\ \mid\ \text{'('}\ \langle\text{bag\_elm}\rangle\ (\ \text{','}\ \langle\text{bag\_elm}\rangle\ )*\ \text{')'} \\
\text{bag\_elm} & ::= & \langle\text{int\_expr}\rangle \mid \langle\text{float\_expr}\rangle \mid \langle\text{boolean\_expr}\rangle \mid \\
 & & \langle\text{string\_expr}\rangle \mid \langle\text{ipaddr\_expr}\rangle \mid \langle\text{set\_expr}\rangle \mid \\
 & & \langle\text{seq\_expr}\rangle \mid \langle\text{bag\_expr}\rangle
\end{array}
\tag{4.12}
$$

The following code defines a bag object representing a bank account. When a bag is created, its objects are first instantiated and initialised using the specified values.

```
acc1 : Account = bag {
  name : string;
  number : string;
  balance : float;
} := ("John Doe","X123-456",1000.00);
```

Different objects from the same type but with different values can be created using the type identifier. The following code shows how to create a new object of the same `Account` bag.

```
acc2 : Account := ("Mary Down","Z000-789",1500.00);
```

A bag defines a locality for its objects. They are not accessed outside the bag without qualifying them with the bag's identifier. This is done using the dot ('.') operator. For instance, the expression `acc1.name` evaluates to `"John Doe"`.

The containment relationship between a bag and its element can be optionally made dynamic. This is denoted by the **dynamic** keyword. A dynamic containment is like reserving a slot for an element. The element can be attached or detached from the bag dynamically. A dynamic containment does not automatically load the specified object. Loading and unloading should use the `attach` and `detach` functions, respectively. Loading is successful if the object is defined and it has not been previously loaded yet. Unloading is successful if the object is in the bag when the operation is performed. Section 4.5.2 illustrates the usage of this feature.

With the dynamic loading and unloading, it is not possible to trace the containment status of an element object at run-time, because such an object can be loaded or unloaded at any time. The dynamic feature is mainly designed to implement the vehicle metaphor of the Mocha modeling paradigm.

Function

The `function` type is designed to represent a functionality. A function object can be regarded as opaque container for a series of algorithmic steps representing an action. This means the algorithmic notion of a function is not visible at the mobility level. A function object can be activated, which means its action is executed. The definition of a function is governed by the following syntax.

$$
\begin{array}{rcl}
\text{function\_def} & ::= & \langle \text{object\_id} \rangle \text{ ' : ' } \langle \text{func\_typedef} \rangle \text{ ';'} \\
\text{func\_typedef} & ::= & \text{([} \langle \text{type\_id} \rangle \text{ ' = '] } \textbf{function} \text{ '(' [} \langle \text{f\_params} \rangle \text{] ')'} \\
 & & \text{-> } \langle \text{type} \rangle \text{)} \\
 & | & \langle \text{type\_id} \rangle \\
\text{f\_params} & ::= & \langle \text{f\_param} \rangle \text{ (',' } \langle \text{f\_param} \rangle \text{) *} \\
\text{f\_param} & ::= & \langle \text{object\_id} \rangle \text{ ' : ' } \langle \text{type} \rangle \\
\text{type} & ::= & \textbf{int} \mid \textbf{float} \mid \textbf{string} \mid \textbf{boolean} \mid \textbf{ipaddr} \mid \\
 & & \langle \text{type\_id} \rangle
\end{array}
\tag{4.13}
$$

A function type is determined by its parameters and its body that defines the action. A function object is different from other types of object as it does not have any value. When executed, a function always return an object whose type is specified in the definition. This object can be assigned to a recipient object using an assignment statement. A non-returning function can be represented by one that returns a `void` object. The following code shows an example that defines a function object `fsq` that takes one integer object argument and returns an object with an integer value.

```
fsq : squareF = function (n : int) -> int
```

If the returned object is of a user-defined type, the type can be defined with the help of a dummy object. Parameter definition can also take advantage of this feature, as shown in the following example.

```
_ : intSeq = seq int;

_ : aBag = bag {
  x : string;
  y : int;
};

f1 : function (s : intSeq) -> aBag
```

In the above example, dummy objects are used to facilitate the definition of `intSeq` and `aBag` types that are required in the function definition.

One may observe that a function's body containing its algorithmic specification is not specified in a definition. The purpose of the omission is to completely isolate the functional abstraction of a function object from mobility specification. In this situation, a function object acts as a black-box representation for the functionality. A function definition therefore denotes the existence a functionality that can be manipulated and activated at the mobility level, but its specification is given elsewhere.

Once a function object is defined, it can be called to execute the action. A function call is made by specifying the object identifier and the required parameters, if any. The syntax is given as follows.

$$
\begin{aligned}
\text{func\_call\_expr} &::= \langle\text{object\_id}\rangle\ \text{'('}\ [\langle\text{params}\rangle]\ \text{')'} \qquad (4.14)\\
\text{params} &::= \langle\text{param}\rangle\ (\ \text{','}\ \langle\text{param}\rangle\ )*\\
\text{param} &::= \langle\text{object\_id}\rangle \mid \langle\text{literals}\rangle\\
\text{literals} &::= \langle\text{integer\_literal}\rangle \mid \langle\text{float\_literal}\rangle \mid \langle\text{string\_literal}\rangle\\
&\quad \mid \langle\text{boolean\_literal}\rangle \mid \langle\text{ipaddr\_literal}\rangle
\end{aligned}
$$

Two distinct function objects can share the same definition. This can be achieved by creating a new function object using an existing function definition. Two function objects are distinct, although they refer to the same functionality. The following example illustrates this.

```
square1 : squareF = function (n : int) -> int
int1 : int := square1(10);
square2 : squareF;              # same as square1
int2 : int := square2(10);      # has the same value as int1
```

In addition to user-defined functions, Mocha has several predefined functions. As with other functions, they are abstract functions in which the details are left to the implementation language. Mocha only defines the generic semantics of these functions.

- The `attach(bag,object)` function.

  The function attaches an object to a bag in its dynamic containment. It is used to denote the **in** capability described in Section 3.4.5. The `object` object must have been dynamically allocated. It must have not been previously attached for the operation to be successful. The function returns the attached object upon a success, void otherwise.

- The `detach(bag,object)` function.

  The function does the opposite of `attach`. It denotes the **out** capability (see Section 3.4.5). It removes an object that was dynamically attached. It produces no effect on non-dynamic objects. It returns the object upon a successful removal, void otherwise.

- The `now(mob_obj,context)` function.

  This is a contextual function. It returns an object with a value representing the current context value of the `mob_obj` vehicle that belongs to a specific context denoted by `context`. The semantics only allows it to be used in a context (`on`) statement.

The function construct is designed mainly to represent task-oriented actions (i.e., those that perform the tasks to solve a problem). Context-oriented functionality (i.e., the one that has direct relevance to mobility) are not strongly supported. Mocha does not encourage developers to work with

excessive algorithmic abstractions at the mobility level. They can still specify actions, though, but there is no specific linguistic construct is designed to represent their abstraction (i.e., it is not possible to define a function at the mobility level).

4.4.4 Non-Context Statements

Non-context statements cover traditional statements commonly found in programming languages. They are designed to accommodate the need for algorithmic representations at the mobility level (e.g., to specify context-oriented functionality). They are only visible in places where a context-oriented action is required. An exception of this rule is assignment statements, which can be used in object definitions to assign an initial value for a newly defined object. The rule implies that it is not allowed to have a non-context statement as a top-level construct.

A non-context statement represents a unit of execution of an implementation program. Collectively, a series of non-context statements exhibit a sequential execution behaviour. The order of execution of the statements is defined by the order of their specification. This behaviour, however, is confined to a limited scope under the control of a state-based execution model.

There are three types of non-context statements, namely assignment, function call, and control statements. They can be grouped into a block of statements as well.

$$
\begin{aligned}
\text{nc\_stmt} \quad &::= \quad (\ \langle \text{assignment\_stmt} \rangle \\
&\quad\ \mid \quad \langle \text{function\_call\_stmt} \rangle \\
&\quad\ \mid \quad \langle \text{control\_stmt} \rangle\ )\ \text{';'} \\
&\quad\ \mid \quad \langle \text{block\_nc\_stmt} \rangle \\
\text{block\_nc\_stmt} \quad &::= \quad \text{'\{'}\ (\ \langle \text{nc\_stmt} \rangle\ )*\ \text{'\}'}
\end{aligned}
\tag{4.15}
$$

Assignment statements are used to assign a value to an object. A value returned as the result of an expression is assigned to the object specified on the left side of the ':=' symbol. Mocha imposes type equivalence in an assignment statement, therefore an expression has to yield a value with a type matching to the that of the object. The syntax of an assignment statement is as follows.

$$\begin{aligned} \text{assignment\_stmt} &::= \langle\text{object\_id}\rangle \text{ ' := ' } \langle\text{expression}\rangle \qquad (4.16)\\ \text{expression} &::= \langle\text{int\_expr}\rangle \mid \langle\text{float\_expr}\rangle \mid \langle\text{string\_expr}\rangle \mid \\ &\quad \langle\text{boolean\_expr}\rangle \mid \langle\text{ipaddr\_expr}\rangle \mid \\ &\quad \langle\text{seq\_expr}\rangle \mid \langle\text{set\_expr}\rangle \mid \langle\text{bag\_expr}\rangle \mid \\ &\quad \langle\text{func\_call\_expr}\rangle \end{aligned}$$

A function call statement executes the function's action and returns an object with a value, if any. A function call statement is mostly used for functions that do not return an object (i.e., like a procedure call in Pascal). However, there is no restriction on using it for functions with a return object (even if a function returns an object, there is no obligation to explicitly assign it to a recipient object).

$$\text{function\_call\_stmt} ::= \langle\text{function\_call\_expr}\rangle \qquad (4.17)$$

Mocha has control structures similar to those in other programming languages. There are conditional and loop statements implemented by the `if`, `for`, and `while` commands. Their syntax is given in the following definition.

$$\begin{aligned} \text{control\_stmt} &::= \langle\text{if\_stmt}\rangle \mid \langle\text{while\_stmt}\rangle \mid \langle\text{for\_stmt}\rangle \qquad (4.18)\\ \text{if\_stmt} &::= \textbf{if } \text{'('} \langle\text{boolean\_expr}\rangle \text{')'} \langle\text{nc\_stmt}\rangle \\ &\quad [\, \textbf{else } \langle\text{nc\_stmt}\rangle \,] \\ \text{while\_stmt} &::= \textbf{while } \text{'('} \langle\text{boolean\_expr}\rangle \text{')'} \langle\text{nc\_stmt}\rangle \end{aligned}$$

for_stmt ::= **for** '(' [⟨for_init⟩] ';' [⟨comp_expr⟩] ';'
[⟨for_update⟩] ')' ⟨nc_stmt⟩

for_init ::= ⟨object_id⟩ ' := ' ⟨int_expr⟩

for_update ::= ⟨object_id⟩ ' := ' ⟨int_expr⟩

4.4.5 Context Statements

A context statement is a top-level construct and independent from other Mocha constructs. It is used to control mobility through object manipulation based on the concept of contexts. Syntactically, it is similar to a case or switch statement, only the selector is a context expression. Its definition is denoted by the keyword **on**[1].

context_stmt ::= **on** ⟨object_id⟩ (',' ⟨object_id⟩) * (4.19)
of ⟨object_id⟩ (',' ⟨object_id⟩) *
[**use** ⟨seq_expr⟩] ⟨on_body⟩

on_body ::= '{' (⟨on_clause⟩) * '}'

on_clause ::= ⟨select_expr⟩ ' : ' '{' (⟨nc_stmt⟩) * '}'

select_expr ::= ⟨bool_term⟩ (⟨log_op⟩ ⟨bool_term⟩) *

bool_term ::= ⟨mob_st_expr⟩ | ⟨bool_expr⟩

mob_st_expr ::= ['!'] ⟨object_id⟩ (**sync** ⟨object_id⟩) *
['!'] (**before** | **at** | **after**) ⟨context_val⟩

context_val ::= [⟨context_id⟩ '/'] (' * ' | ⟨integer_literal⟩ |
⟨float_literal⟩⟨string_literal⟩ | ⟨boolean_literal⟩ |
⟨ipaddr_literal⟩ | ⟨bag_expr⟩ | ⟨range_expr⟩)
'/' (' * ' | ⟨int_expr⟩)]

context_id ::= ⟨object_id⟩ ('.' ⟨object_id⟩) *

[1]The statement is also called the `on` statement.

Linguistic Semantics

A context statement implements Mocha's computation model as described by Definition 3.24 in Section 3.4.7. Its explanation starts from the `on` clause. An ⟨object_id⟩ tag in this clause defines the object of context manipulation. The `of` clause defines one or more context domains. The combination of the two clauses defines the awareness of the object of the contexts represented by the context domains. The `use` clause optionally assign a sequential property to enable a deterministic program execution.

The ⟨on_body⟩ is the place where context-awareness clauses are specified using a format similar to a case or switch statement. The left-hand part of a clause is a selector expression. In addition to ordinary boolean expressions, a selector allows the use of *mobility state expressions* (denoted by the ⟨mob_st_expr⟩ tag). A mobility state expression is a boolean expression that works with context operators **before**, **at**, and **after** (see Section 3.4.2). Evaluation of a mobility state expression is performed by contextually comparing the specified object, which has to be defined in the `on` clause, with a context state. If an evaluation produces a true result, the statements in the right-hand part of the selector expression will be executed.

The right-hand part of the selector expression is the only place where algorithmic specification is allowed in a Mocha program. Non-context statements specified in this part are executed in an order specified by their specification order. Mocha does not provide any means to encapsulate this part into a single abstraction (e.g., to encapsulate the statements into a function).

The **sync** modifier in a mobility state expression indicates a synchronisation using the specified context state. It is used to define a rendezvous between the participating objects (see Section 4.5.7).

The context state used in a mobility state expression is denoted by the ⟨context_val⟩ tag. It must come from a context domain specified in the

`of` clause. The optional ⟨context_id⟩ tag serves as a context identifier in the case of the presence of multiple contexts. If there is only one context defined in the `of` clause, it may be left empty. If present, a context identifier is followed by the specification of a context value. A '*' symbol in this part indicates a match with any value from the corresponding set. The last part is an occurrence counter. A '*' symbol indicates a match to any number of occurrence.

The expression !⟨object_id⟩⟨op⟩⟨context_val⟩ needs some explanation. As mentioned by Statement 3.18 in Section 3.4.4, it means either the object or the contextual association between the object and the context referred by the context value does not exist. In most cases, the falsification is caused by the latter factor. This happens when the object temporarily experiences a situation that is not expressable by the context (i.e., the context value associated with this situation is not defined in the context domain).

The ! operator in the expression ⟨object_id⟩!⟨op⟩⟨context_val⟩ refers to the mobility state. It falsifies the specified mobility state and toggles the boolean values of the other corresponding mobility states.

The linguistic semantics of a context statement can be summarised as follows. The objects defined in the `on` clause are assigned vehicle semantics. Its mobility is represented by its context-awareness specified in the body of the statement. The same mechanism is also used to trigger the execution of the actions associated with the vehicle. The following example models the mobility of a salesman who tries to sell his goods in every place he visits.

```
salesman : bag {
  sell : function () -> void;
};

location_set : set string := {"London","Sydney","Tokyo"};

on salesman of location_set {
  salesman at */* : { salesman.sell(); }
}
```

Execution Semantics

A context statement models an execution entry point of an executable program. There can be more than one context statement in a program, and each statement is executed independently of the other. This has an effect of creating concurrent program execution.

When executed, a context statement behaves like a loop statement. It loops over the mobility states, looking for a matched clause. In every iteration, each clause is evaluated, possibly triggering an action execution. There can be more than one match in an evaluation round, and actions can be executed concurrently. Clauses are evaluated in an arbitrary order.

The difference between an `on` loop and a traditional loop is that its iteration is not driven by the statement itself. Instead it is driven by the context associated with the statement. The loop moves to the next round if it is triggered by a context state change that can be captured by the loop. A state change can be captured if its corresponding context value is listed in the set representing the context domain. Only representable context state changes can be used to trigger an action.

In a program with multiple context statements, mutually exclusive statements will run independently. If a particular vehicle is present in more than one statement, then a state change in a particular context experienced by the vehicle is reflected in all statements in which the vehicle is involved.

```
on M1 of Context1 {
  M1 at */* : { M1.run1(); }
}

on M1 of Context2 {
  M1 at */* : { M1.run2(); }
}
```

Context state changes in both `Context1` and `Context2` affect `M1` in both statements. A similar effect is produced by the following specification.

```
on M1 of Context1, Context2 {
  M1 at Context1 /*/* : { M1.run1(); }
  M1 at Context2 /*/* : { M1.run2(); }
}
```

The difference is that a program implementing the above specification only requires one execution thread.

Another unique characteristics of a context statement is its *binding* semantics to a contextual situation. *Contextual* binding is provided by a set element which represents a context value. When a mobility state evaluates to true, the corresponding statements will be executed, and the computation is contextually bound to the state.

Contextual binding is most useful for contexts with spatial semantics. It is an expressive way to say that a computation is *physically* bound to a spatial entity. Given the ability of Mocha to express various kinds of spatial context, this allows modeling of computation that is bound to, for example, a machine, a cluster of machines, or a geographical location pointed by a GPS coordinate. The following example illustrates the concept.

```
agent1 : function () -> void;

localnet : set ipaddr := {192.168.10.10~20};

on agent1 of localnet {
  agent1 at 192.168.10.15/* : { agent1(); }
}
```

Any time the `agent1` function object is being at the machine with IP address 192.168.10.15, its functionality is activated. The execution will be bound to the specified machine. This means the thread of execution is migrated to that location, and the link to the original location is no longer maintained. The new place becomes the new locality for the computation.

The previous example also shows another feature of Mocha. It allows mobility feature to be embedded to any type of object. In the example, a

mobile property is assigned to a function, and the `agent1` object represents both the mobility and functionality aspects. The same technique could be used to make a passive data (e.g., a database) mobile. In this case, no action is activated.

4.5 Working with Mocha

This section explains how mobility control is specified using Mocha. It explains the semantics of Mocha constructs in more detail through examples. Examples are given using the space ship scenario presented in Section 3.3.1. To recall, it describes a space ship carrying an astronaut who performs some analysis task on samples collected by a robot from the planets visited by the space ship. The space ship is equipped with an autopilot that can make necessary adjustments if there are weather problems during the journey.

4.5.1 Functionality Representation

A functionality is implemented by a function object. In many cases, a functionality has a unidirectional characteristic. This situation occurs when a task needs to be performed in other places by migrating it, and there is no requirement to retrieve the result. It could be implemented by defining and calling a function and disregarding its return object. For example, the following definition creates a reference to a function that represents the sample analysis task performed by the astronaut. The function accepts one argument and returns no object.

```
sample : Sample = bag {
           param1 : string;
           param2 : int;
           ...
};

analysis :  function (p : Sample) -> void;
```

When the `analysis` object is invoked in an `on` statement, its functionality is activated at the locations specified in the statement.

A bidirectional characteristic is exhibited when a function has some relevance to mobility. This is shown by a return object that will be used at the mobility level. This situation is exemplified by the sample collection task. The robot does this task, returning with some sample collection. The sample can then be used by the analysis function described previously.

```
sampColl : function () -> Sample;
```

The `sampColl` is an example of a task-oriented function. The function is directly associated with the problem to be solved. The autopilot function, on the other hand, is an example of a context-oriented function. It represents a functionality that is closely related to the mobility of the space ship. The following example shows the capability of the autopilot to slow down, accelerate, and change the destination of the space ship.

```
accel : function (delta : float) -> void;
slowing : function (delta : float) -> void;
changedest : function (dest : string) -> void;
```

The previous examples show the way Mocha handles the functionality aspect. It provides a uniform way to represent a functionality possessed by any type of mobile entity. All are treated the same, regardless of their purpose and who owns them. This feature offers a big advantage for mobility control, as it is directly applicable in both code-oriented and user-oriented mobility), or even in a mixed situation where both mobility types are present.

4.5.2 Mobile Entity Representation

Mobility is not associated with a Mocha construct. Any construct can be made mobile by using it in a context statement. However, a mobile entity is conveniently modeled by a bag. The structural characteristic of a bag is useful

to abstract the physical notion of a mobile entity, as shown by the following example that models the robot with the sample collection functionality.

```
aRobot : Robot = bag {
  sampColl : function () -> Sample;
};
```

The elements of the `aRobot` bag moves together as the bag moves. This feature allows encapsulation of functionality and its supplementary data. At the implementaton level, it defines the scope of the migrating objects. All objects in the scope are local to each other and move together as a whole.

It should be noted, however, that a mobile entity does not have to be represented by a bag. If the mobility of the functionality is more important than its physical notion, then a bag is not necessary. For example, the `sampColl` object can be made mobile by itself.

A bag, however, is required to represent multiple hierarchical vehicles. Multiple hierarchical mobility (i.e., migration of multiple mobile entities that form a hierarchy), as shown by the space ship and robot in the scenario in Section 3.3.1, can be represented by a hierarchical bag structure. The dynamic membership that allows dynamic loading and unloading makes it easy to mimic the mobile characteristic of a vehicle.

```
aRobot : Robot = bag {
  sampColl : function () -> Sample;
};

ship : Ship = bag {
  sample : Sample = bag {
             param1 : string;
             ...
  };

  analysis : function (p : Sample) -> void;

  # the robot can leave and enter the space ship dynamically
  dynamic robot : Robot;
} := (_,_,attach(self,aRobot));
```

Some parts of the code are worth mentioning. The definition of the `sample` object and the `analysis` function are made *inside* the `ship` object to get the effect of local scoping. The field objects become statically part of the parent object, and their existence is dependent on the parent's. They do not need to be initialised, so dummy objects are used in the initialisation part. Static containment as shown in the example can be used to model an object that becomes a fixed part of a bigger object.

The `robot` object, on the other hand, will leave the space ship and enter it again at some point. The `dynamic` modifier satisfies this relationship. The definition of the robot is made outside of the definition of the space ship, because it is necessary to make the robot visible outside the ship (e.g., when the robot leaves the space ship and is subject to an independent mobility control mechanism). In this scheme, the initialisation of the ship explicitly attaches the robot into the space ship.

4.5.3 Context Specification

A vital element in mobility control is the context on which context-based manipulation is based. Contexts are represented using sets, with set members representing context values. Context values can be enumerated individually, specified using ranges, or defined implicitly. The following example illustrates possible ways of defining a context.

```
socialCtx : set string := {"jim","emily","ruth","bob"};
labSubnet : set ipaddr := {[190.168.100.1~254]};
netSpeed  : set float  := {float};
```

The first example defines a context of persons that are in the vicinity (e.g., those who are logging on the same machine as the user associated with the vehicle using this context). The second definition uses a range set to define a set of IP address. The last line in the example defines an infinite set (e.g., to represent a network speed context).

Mocha also facilitates the definition of contexts formed by several primitive abstractions. Such contexts can be represented using bags. For example, a *room* context is constructed from the abstractions of the room number and the machine located in a room. This can be specified as follows.

```
room1 : Room = bag {
  number : int;
  host : ipaddr;
} := (1,190.168.10.10);

room2 : Room := (2,190.168.10.11);
...
room5 : Room := (5,190.168.10.15);

roomCtx : set Room := {room1, room2, room3, room4, room5};
```

It should be noted that the previous set definitions do not bear the semantics of a context yet. They become context representations when they are used in a context statement.

4.5.4 Context-Awareness and Action Activation

When an object is used in a context statement, it is bound to the context specified in the statement. The object can listen to any change of a context state, and effectively become a vehicle. So from the previous space ship example, the behaviour of the space ship can be specified as follows.

```
DestSet   : set string := {"Mars","Pluto","Saturn",
                           "Venus","Jupiter","Uranus"};
on ship of DestSet {
  ship at */* : { ship.analysis(sample); }
}
```

In the example code, the space ship is made aware of any change in the context `DestSet` by binding it to the context statement. Every time the ship changes its location, the change is captured in the statement and reflected by the true value of the `ship at */*` expression. The term `*/*` denotes a match to any context value at any occurrence.

The previous example shows how a context statement binds two concepts into a single operational model: assignment of mobility semantics to an object through a context-awareness mechanism, and activation of functionalities using the same mechanism. Combined with the genericity of the concepts of object and function (i.e., they can represent any mobile entity and functionality that belong to it), this model is powerful enough to describe any mobile system.

Mocha is also capable to model deterministic systems. Many mobile applications use a predetermined pattern of movement, so it is desirable to have control over such a pattern. Mocha uses a mechanism that abstracts the sequential property of a sequence to facilitate this.

```
DestSet   : set string := {"Mars","Pluto","Saturn",
                           "Venus","Jupiter","Uranus"};

vSeq : seq string := ["Mars","Jupiter","Saturn"];

on ship of DestSet use vSeq {
  # anywhere, at any occurrence
  ship at */* : { ship.analysis(sample); }
}
```

The `vSeq` object acts as an ordering guide for the migration of the space ship. Equipped with a sequence, the context statement behaves like a `for` loop. Starting from the first element, each element is fetched in each iteration where the mobility state clauses are evaluated. Note that the space ship only visits three planets instead of six, even though a `*/*` is specified.

A vehicle may listen to more than one context. For example, the space ship may listen to the location and weather contexts. The following code shows the specification.

```
DestSet   : set string := {"Mars","Pluto","Saturn",
                           "Venus","Jupiter","Uranus"};

WeatherIndex : set int := {0~30};
```

```
on ship of DestSet, WeatherIntensity {
  ship at DestSet/*/* :
    { ship.analysis(); }
  ship at WeatherIndex/[0~10]/* :
    { ship.changedest(alt_dest); }
  ship at WeatherIndex/[11~20]/* :
    { ship.slowing(minspeed); }
  ship at WeatherIndex/[21~30]/* :
    { ship.accel(maxspeed); }
}
```

Since evaluation of the clauses are done independently, it is possible to have more than one clause with a true evaluation result at the same time. In general, if multiple contexts are present, evaluation of mobility states is instantaneously triggered upon an occurrence of a context state change caused by any one of the contexts. In the above specification, there is nothing to prevent the autopilot from working when the analysis is being performed. If a more strict situation is required, the code has to be modified as follows.

```
on ship of DestSet, WeatherIntensity {
  ship at DestSet/*/* : { ship.analysis(); }

  ship at WeatherIndex/[0~10]/* && (! ship at DestSet/*/*) :
    { ship.autopilot("detour"); }
  ...
}
```

The expression `!ship at DestSet/*/*` denotes a situation where the space ship is losing its association with the context specified by the `DestSet`. This happens when the space ship is in the middle of a journey between planets. As this situation is not expressable by the planet context, the ship is said to be out of that context during this time. The whole context value expression describes the space ship being in the middle of a journey and experiencing a bad weather.

In general, falsification of an association with a context is used if it is required to express a *disconnected* context-awareness.

4.5.5 Dynamic Structuring Mechanism

The interaction between the robot and the space ship is shown in the following example. It makes use of the `detach` and `attach` functions.

```
DestSet   : set string := {"MarsStation","PlutoStation",
                           "VenusStation","JupiterStation"};

RobotDest : set string := {"spot1","spot2","spot3"} + DestSet;

on robot, ship of DestSet, RobotDest {
  ship at DestSet/*/* : {
    ship.analysis(sample);
    detach(ship,robot);
  }

  (robot at RobotDest/*/1) &&
  (now(robot,RobotDest) == now(ship,DestSet)) : {
    attach(ship,robot);     # enter the space ship
  }

  robot at RobotDest/*/1 : { sample := robot.sampColl(); }
}
```

The activity pattern of the space ship shows that every time the astronaut completes the analysis task, the robot is sent out to collect samples. The robot's mobility is specified in a separate context statement. In every location, it does its job, and when its location is the same as the space ship, it is ordered to enter the space ship. Note how to represent common location of the robot and space ship (i.e., where the robot enters the space ship) using set addition operation. Mocha does not interpret the granularity of the context values of the sets representing the two different location abstractions. In this example, they are assumed to have a comparable granularity to allow the notion of "being at the same place" to be represented. The `now` function returns the current context value for a vehicle, and is used to check whether the robot and the space ship are at the same location.

The previous example illustrates how to detach and attach a vehicle from another vehicle to model multiple mobility with some dependent char-

acteristics. When the robot is being contained by the space ship, it is subject to the space ship's mobility rule described by the first context statement (i.e., it would move along with the ship to the destination planets). The example, however, does not show a required condition that the space ship waits for the robot before moving to the next planet. The specification allows, for example, the ship to continue to another planet while the robot is exploring the current planet, and return back later to pick up the robot. To solve the problem, a strict rendezvous is required, and this will be explained in Section 4.5.7.

The example also shows how to communicate different tasks using data objects. The `sample` object is used to represent a logical task sequence performed by the `sampColl` and `analysis` functions. Note that Mocha does not go into the detail of how the task sequence is carried out. The `sample` object only indicates that there is an *interface* between the two functions.

4.5.6 Task Composition

For mobile applications with non-trivial mobility, program composition is crucial. The problem is how a problem involving mobility can be decomposed into smaller subproblems, to which problem-solving components can be designed and implemented, and then composed to form a single program.

Task composition is done through context statements. A context statement is designed to provide a single abstraction for a task involving mobility. The functional aspect of the task is defined by the function elements encapsulated in vehicles. A vehicle wraps functionalities that are logically inseparable. Contexts, on the other side, define the mobility aspect. A context statement binds the two aspects into a single abstraction.

Mocha offers two different ways of composing tasks. The first approach is *task-based* composition. In a program with multiple mobile functionalities

with different agendas, tasks can be composed along the functions. The goal is to achieve cohesive abstractions of functional modules. The following example illustrates the concept. For instance, a program needs to perform two independent tasks, T1 and T2, which are wrapped into two vehicles, V1 and V2, respectively. T1 needs to be executed at A, B, C,and D, while T2 is performed at B, C, and D. The program can be composed as follows.

```
Dest1 : set string := {"A","B","C","D"};
Dest2 : set string := Dest1 - {"A"};   # means B,C, and D

on V1 of Dest1 {
  V1 at */1 : { V1.T1(); }
}
on V2 of Dest2 {
  V2 at */1 : { V2.T2(); }
}
```

In general, multiple context statements with a single vehicle gives the best description of *what* is going to be performed by the mobile entity.

The above example also shows how parallel and independent mobility is specified. The two context statements are mutually exclusive because they work with unique vehicles which have not been used anywhere else. In the above example, there will be two separate roaming threads running independently, and there is no execution dependency between them.

In a real-world situation, functional cohesion may not be the only important factor in mobile application development. For example, efficiency is often crucial in a distributed computing environment. In some cases it may be desirable to favour some implementation aspects rather than to program readability. In the previous example, realising that both tasks are performed in some common destinations, it may be preferred to have a single roaming thread, probably sacrificing functional modularity of the specification. To accommodate this, Mocha provides an alternative to the first composition approach. This approach tends to favour *contextual* composition, as shown in the following code.

```
Dest : set Loc := {"A","B","C","D"};

on V1,V2 of Dest {
  V1 at */* : { V1.T1(); }
  ((V2 at B/*) || (V2 at C/*) || (V2 at D/*)) : { V2.T2(); }
}
```

The program is shorter, but it is more difficult to understand. It is also more efficient, because it only has one thread. Although vehicle movement is independent from each other (i.e., they go to different locations), their processing (i.e., evaluation of mobility states) are performed by the same thread. Such a centralised control opens the possibility for efficiency tuning.

4.5.7 Rendezvous

Mobile systems with multiple mobility but no interaction between the mobile entities are very rare. In most cases some form of communication takes place among these entities. As explained in Section 3.3.6, such an interaction is facilitated by the *rendezvous* mechanism. To recall, a rendezvous is a meeting between two vehicles so that their underlying functions can perform some interaction.

Similar to an action, a rendezvous is bound to the context mechanism. A rendezvous between two vehicles can happen only if they are *contextually* local to each other. It means they have to listen to the same context and responds to the same mobility states. A rendezvous is triggered if the shared mobility states hold.

Mocha uses the `on` construct, combined with the `sync` modifier, to specify a rendezvous. A context statement defines the vehicle objects that propose the meeting and the mobility state on which the meeting is based. The `sync` modifier tells the given vehicles to perform some synchronisation to do some interaction. The following example shows a rendezvous specification.

```
Dest : set string := {"A","B","C","D"};
```

```
on V1,V2 of Dest {
  V1 at */* : { V1.T1(); }    # clause for V1
  (V2 at "B"/*  || V2 at "C"/* || V3 at "D"/*) :  # clause for V2
    { V2.T2(); }

  V1 sync V2 at "C"/* : { .... }   # meet at C
}
```

The semantics of a rendezvous is as follows. The `sync` modifier affects the evaluation of the mobility state clauses. If a vehicle is `sync`ed to a mobility state, it will be "locked" to that mobility state, and will not be affected by a context state change until its corresponding statements have been completely executed. In the above code, if either `V1` or `V2` is experiencing the state denoted by the context value `"C"`, it remains in this state until the other vehicle comes to the same state. Once this happen, the corresponding statements are executed. Upon the completion of the execution, the locks are released and the vehicles become aware of context state changes again.

The correct interaction behaviour of the space ship and the robot can be modeled using the rendezvous mechanism. The task sequence can now be correctly specified. When the ship arrives at a planet, the analysis task is executed, followed by the despatching of the robot. Next, the ship has to wait for the robot to enter it before proceeding to the next planet.

```
DestSet   : set string := {"MarsStation","PlutoStation",
                           "VenusStation","JupiterStation"};

RobotDest : set string := {"spot1","spot2","spot3"} + DestSet;

on robot, ship of DestSet, RobotDest {
  ship at DestSet/*/* : {
    ship.analysis();
    detach(ship,robot);
  }

  # the robot is at the same location as the space ship
  ship sync robot at DestSet/*/* : {
    attach(ship,robot);     # enter the space ship
  }
```

```
  robot at RobotDest /*/1 : { robot.sampColl(); }
}
```

Note how the timing constraint is handled in the previous specification. The robot has to be given some time to collect samples before returning to the space ship. Therefore there has to be a time lapse between a call to `detach` and the following call to `attach`. Recall that `detach` implements the **out** capability. Calling it makes the robot atomically leave its current position where the ship is located. This prevents immediate activation of the second clause, allowing the robot to collect samples. The second clause would only be activated when the robot returns to the space ship's location. It implies that the robot has performed sample collection.

Recall that there are two types of rendezvous (see Section 3.4.8). Strict rendezvous does not allow context state change to happen before both parties finish their interaction, while loose rendezvous allows it. Most interactions require strict rendezvous, and this is supported by default by Mocha and reflected in the semantics of a context statement. It is shown by the locking mechanism of the `sync` operation that prevents a vehicle from changing its state until the actions are completely executed.

Loose rendezvous, on the other hand, allows context state changes to happen during a rendezvous. It does not require specific support. The prerequisite for a rendezvous, i.e., the contextual locality, can be modeled using the `now` function. Once a rendezvous takes place, there is no need to guard it from a context state change.

4.6 Discussion

The specification and features of Mocha language has been discussed in this chapter. The language is designed as a specification tool for the Mocha modeling paradigm. It can be used to describe a mobile system by defining the

system's components and their relationship. Mobile entities are created by defining and structuring mobility and functional components, and function execution is controled by a context-based mechanism.

The way the behaviour of a mobile system is controled in Mocha is different from that of the traditional approach. Control specification is written in linguistic constructs with high-level, user-oriented semantics, but still following a syntax similar to a programming language. This representation style reflects the intermediary role of the Mocha language.

As a specification tool, Mocha focuses on some aspects that become the highlights of its modeling paradigm. The model's genericity is supported by keeping the language's constructs away from implementation-specific semantics. The syntax and semantics of Mocha also support the principle of separation of concerns. They set a framework which concentrates on the mobility concern and prevents functionality from being specified in a detailed fashion. Finally the relationship between a mobile entity and its physical relation is captured by a context statement. The statement also represents Mocha's uniform mechanism to control any action in the realm of the mobility model.

As a specification language, Mocha has a responsibility to pass the abstractions from the modeling stage to the design and implementation stages. This is required so that the design and implementation of a mobile application can be based on a well-defined system specification.

Language design becomes important in this situation. The design should facilitate gradual abstraction refinement in the design and implementation stages. This goal is achieved by borrowing some design concepts from programming languages and modify them to operate on the Mocha's modeling paradigm. Concepts like objects, types, and statements are commonly used in programming languages, and by adopting them in a model

specification, it should not be difficult to transform the specification into an implementation program. This explains why Mocha excels in modeling mobile applications, compared with other specification languages such as Z [Spi92] and LOTOS [vEVD89]. They are designed without explicit intention to provide assistance in implementing a program based on it. Even though extensions for such specification languages exist (e.g., abstract data typing for LOTOS [QPM+93]), they still cannot easily bridge the abstraction gap between modeling and implementation levels.

Mocha is also compatible with other modeling tools. The dualism between functionality and mobility aspects can be used to achieve an integrated view of mobile application modeling. Modeling is done in a two-tiered fashion, reflecting the relationship between the two aspects. Mocha handles modeling at the mobility level, while tools such as DFD specifies the process- and data-oriented abstractions at the lower level. For example, given the following Mocha specification,

```
DestSet   : set string := {"Mars","Pluto","Saturn",
                           "Venus","Jupiter","Uranus"};

on ship of DestSet {
  ship at */* : { ship.analysis(sample); }
}
```

the `analysis` function can be described further by descending to the functional level. A possible description of this function is shown in Figure 4.1.

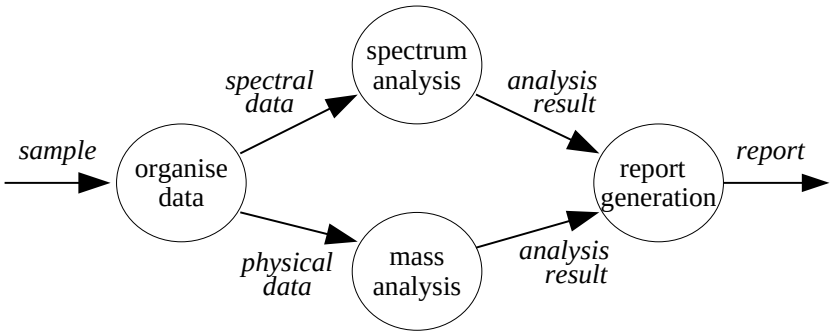

Figure 4.1: A DFD for the `analysis` function

The above modeling strategy suggests the *multiple facets* of application modeling. Different aspects of an application are modeled using different approaches that suit the nature of the aspect. This strategy is employed by UML that provides different modeling views (hence its "Universal" name comes from). It has an advantage of being able to precisely specify the object of modeling, since the view is designed to capture the essential features of the modeled aspect. In this perspective, Mocha's role is to facilitate modeling of the mobility aspect of a mobile application. It does not attempt to be universal, but it can act as a complement to other tools designed to model other aspects of an application.

The next step towards a real software product is the design and implementation stages. The computing environment where an implementation will be carried out has to support some implementation requirements listed as follows.

- The ability to represent mobile entities.
- The ability to represent functionalities.
- The ability to capture and represent environmental elements.
- The ability to represent mobility states and perform evaluation on them, which is combined with the ability to execute functionalities.

The most common way in implementing a Mocha specification is to follow the traditional programming course in which a model is implemented in a programming environment supported by a programming language and other tools such as libraries and debuggers. A programming language usually has a good abstraction handling, so Mocha's constructs could be implemented easily. Mocha's context-based execution mechanism can also be implemented, or at least emulated, using the language's execution model. In

this development course, specification code is transformed into its equivalent form using the programming language's constructs. Mobility components resulting from the transformation process are composed with components that build the functional aspects to create an executable program.

A mobile application does not have to be implemented in a programming environment. Operating system shell is another possible alternative for a development environment. To some extent, Linux (and Unix in general) tools and their programming philosophy provide the required implementation support for certain types of applications.

The next two chapters deal with the implementation aspect of Mocha specification. Chapter 5 discusses implementation following a traditional programming course. The Java programming environment is used as the development environment. Chapter 6 discusses an implementation on the Linux shell environment. The purpose of the research described in these chapters is to evaluate the usability of Mocha's modeling paradigm and specification language to assist mobile application development in different computing environments.

Chapter 5

A Programming Framework for Mocha

5.1 Introduction

The Mocha language for mobility control specification has been presented in Chapter 4. In Mocha's framework, mobility control is specified at the system modeling level. The output is a specification program that models the behaviour of a mobile system. System description is represented using high-level linguistic constructs that abstract the notion of mobile entities, the physical environment elements that surround them, and the functionality of the entities.

A specification program has to be implemented to produce an executable program. Mocha constructs has to be translated into program components at a lower level. The components then have to be composed with those representing the functional aspects. This implementation process completes the development of a mobile application (i.e., it produces a tangible software product), and becomes the essence of the research work described in this chapter. The research work also evaluates the development approach proposed in this thesis. It is expected that invaluable lessons can be learned from the experiences gained during the implementation course.

The implementation of Mocha specification is carried out using a framework-based approach. This approach is used because a programming framework is relatively easy to develop, and from a programmer's point of view, it provides a guided assistance so that the programmer needs only to concentrate on a specific scope of his or her implementation task.

The framework described in this chapter uses Java as the target language. The selection is mainly based on practical reasons. Java is a popular programming language which has support for distributed system programming. This feature is most useful to implement location-based computation, which becomes the core part of many existing mobile applications. Java is also backed-up by extensive libraries and tools for working on various application areas. They help programmers save time and effort in coding a program.

It should be noted that the programming framework is developed as a prototype. The prototype does not elaborate all aspects of the specification language, as this can lead to technical difficulties which require substantial amount of programming time and effort. For example, the prototype does not support multiple mobile entities and hence the rendezvous mechanism. Otherwise, it has to deal with management of multiple threads and thread synchronisation that spans over a distributed computing environment, a programming topic that is beyond the scope of this research.

The main issue in developing the programming framework is the process of transformation of a specification program into an executable Java program. It is not a trivial task due to the abstract nature of the semantics of Mocha constructs. This issue is discussed in Section 5.3. Before that, the proposed approach for developing the framework is explained in Section 5.2. The design and implementation of the framework is presented in Section 5.4. Finally this chapter is concluded by Section 5.5 that presents the lessons learned from the development experience.

5.2 From Specification to Implementation: The Framework Approach

The essential problem of translating a Mocha specification to a Java program is how to transform Mocha's object model to that of Java, and to implement the context mechanism embedded in a context statement. Object model transformation is required because the semantics of Mocha's object model and Java's are different due to the difference of their operational level. Implementation of Mocha context statements is necessary because Java does not provide intrinsic support for Mocha's context-based execution model.

Implementation of a Mocha specification requires more than just a transformation of its object model and implementation of its context-based mechanism. As mentioned in Chapter 4, Mocha retains some degree of abstractness, so some details required in its implementation are still missing. The details are missing because it leaves the semantics of some constructs open to different methods of implementation. For example, consider the following Mocha set definition representing a context.

```
mycontext : set string = {"stall","low","medium","high"};
```

The definition states that the object `mycontext` is a context which has four distinct context values expressed as strings. The information is not sufficient to build a complete Java program. It does not specify, for example, the meaning of each string representing a context value (e.g., what is the meaning of a "low" ?). Furthermore, it does not tell how to get a context value (e.g., to monitor actual physical measurement and to convert it to an appropriate category). This gap must be filled during an implementation work to produce a complete executable program.

The above situation can be approached using the *software framework* technique. A framework is a set of cooperating modules that make

up a reusable design for a specific software domain [GHJV95]. A framework provides architectural guidance by partitioning a software design into some structural parts implemented by its modules. It defines the responsibilities of the modules and their collaboration. Therefore a framework supplies an infrastructure which dictates the architecture of an application, and programmers use it by filling some application-specific functionality.

A programming framework for Mocha can be useful to help programmers build a Java program that implements a Mocha specification. The framework creates the overall structure of the application built from core mobility components, and programmers provide the details of the components by defining their operational properties. The advantage of this approach is that it frees programmers from writing implementation code from scratch. This bottom-up practice is discouraged, because it makes it difficult for programmers to have a complete picture of the application. The framework, on the other hand, preserves Mocha's conceptual view of the application. It carries over the high-level abstractions of a model specification and passes them to the implementation level in a smooth transition.

The scope of the framework covers two main areas: the representation of mobile entities and the implementation of the context-based execution mechanism. Mobile entities are represented by creating a set of object structures. The object structures implements the structural property of vehicles and emulates a context-awareness mechanism required by the context-based execution mechanism.

The implementation of the context-based execution model is more challenging, because it has to mimic the non-procedural nature of the state-based execution model. The framework makes use of the event-based notification system to emulate the asynchronous and non-deterministic interactions between a vehicle and the context-awareness mechanism. Programming mobility state evaluation is also another challenge, since it is not trivial to

represent the high-level semantics of contexts and perform contextual comparison on them. The framework helps by providing some basic semantics for some operations required in this activity.

Using the programming framework, the implementation a Mocha specification can be simplified into the following steps. They are exactly the same as those described in Section 4.2, indicating a consistent development process.

1. Define the functions.
2. Define the vehicle for the functions.
3. Specify the mobility control.

These steps are carried out as normal object-oriented programming, freeing developers from working with concepts that are incompatible with their programming customs.

5.3 Object Model Transformation

Before discussing the design and implementation of the framework, it is necessary to mention some guidelines that explain in general how transformation from a Mocha code to a Java program is conducted. These guidelines are useful to avoid conflicts between user-defined code and the framework's code.

Recall that Mocha is an object-based language with an abstract notion of objects. All entities are represented by abstract objects. It means that an abstract object is free from implementation-specific semantics such as object creation, object references, and thorough type systems. Mocha includes a limited type system for its objects, but it is mainly designed as a guidance for an implementation process, and does not become part of the core modeling abstraction.

Java, on the other hand, is an object-oriented language with its own object model and works with real objects. Apart from the core object model, it has to deal with implementation aspects and has to incorporate design decisions on these aspects into its language design. For example, Java's type system is much more complicated than Mocha's, because it has to cover implementation-specific aspects like object references. These kinds of things make a transformation of Mocha's object model to that of Java non-trivial.

Mocha, however, provides basic information for such a transformation to take place. With the help of the framework, what is required by programmers is a set of guidelines to help them perform the transformation. The rules are given in the following.

1. *Mocha's simple types are mapped to their equivalent type, or to the corresponding Java classes that represent the type.*

   There are some exceptions for this rule. The `ipaddr` type has no equivalence in Java, which uses a string representation to denote an IP address. String is not a simple Java type, so an initialisation means creation of an object of class `String`. The `void` object maps to a null object or void type. Finally, dummy objects are not needed in a Java program, because it is used to represent a situation where a type information is expected instead of an object, which never occurs in Java.

   Some examples of object transformation are shown as follows. The left part shows Mocha definitions, and the right part shows their equivalent form in Java.

```
i : int;                          int i;
f : float := 0.1;                 float f = 0.1;
b : boolean := true;              boolean b = true;
s : string := "Mocha";            String s =
                                    new String("Mocha");
p : ipaddr := 192.168.10.10;      String p = new
                                    String("192.168.10.10");
```

Simple types can also be mapped to objects of the corresponding type. The following Java statements can also be used for the first three corresponding Mocha statements mentioned previously.

```
Integer i;
Float f = new Float(0.1);
Boolean b = new Boolean(true);
```

The decision to use a simple type or a class representation is left to the programmers. The latter is generally required if the objects are used in operations that do not permit simple type representation.

2. *The set and sequence types are mapped to predefined Java classes.* They can be implemented using framework classes that have similar semantics. The `Set` class provides necessary functionality for the set type, most notably the set operations. The important features of the `Set` class are shown below.

```
public class Set {
  public Set(Object[] elm)        // constructor accepting
                                  // an array of object
  public void add(Object elm)     // adds a set element
  public void add(Object[] elm)   // adds an array of elmnts
  public Object get(Object elm)   // gets a set element
  public Object[] get()           // gets set elements
  public Object remove(Object elm)  // removes a set element
  public void clear()             // removes all elements
  public int size()               // gets set size
  public Enumeration elements()   // an enumeration of
                                  // all set elements
  public boolean isEmpty()        // true if set is empty
  public boolean equals(Set s)    // equality operation
  public boolean subsetOf(Set s) // subset operation
  public Set intersection(Set s) // intersection operation
  public Set union(Set s)         // union operation
  public Set difference(Set s)    // difference operation
}
```

Consequently, a set object definition with initial value assignment may have to be implemented in more than one Java statements. This is shown by the following example.

```
set1 : set int := {1,2,3};      Object[] m = {new Integer(1),
                                              new Integer(2),
                                              new Integer(3)};
                                Set set1 = new Set(m);
```

The implementation of the sequence type follows the same approach. It is implemented by the `Seq` class which wraps a single linked-list container.

```
public class Seq {
  public Seq(Object[] elm)        // constructor accepting
                                  // an array of object
  public void add(Object elm)     // adds an element
  public void add(Object[] elm)   // adds an array of elmnts
  public Object get(Object elm)   // gets an element
  public Object get(int i)        // gets an element
  public Object[] get()           // gets all elements
  public Object remove(Object elm)  // removes an element
  public Object remove(int i)     // removes an element
  public void clear()             // removes all elements
  public int size()               // gets sequence size
  public Enumeration elements()   // an enumeration of
                                  // all sequence elements
  public boolean isEmpty()        // true if empty
}
```

A sequence object definition may also have to be implemented in more than one Java statement, as shown in the following example.

```
seq1 : seq int := [1,2,3];      Object[] m = {new Integer(1),
                                              new Integer(2),
                                              new Integer(3)};
                                Seq seq1 = new Seq(m);
```

The implementation of both the set and sequence abstractions are commonly found in third-party libraries. They can be fine-tuned to create the required specification.

3. *Functions are transformed into methods.*

   A Mocha function object is like a template for its implementation. Programmers have to define the details of the functionality in the function's body.

A function object is transformed into a method. While a function is an autonomous object, a method is not. Therefore it is necessary to wrap a method definition with a suitable class. The name of a function object may safely be elected as the method name.

```
f : function (num : int)
    -> float;
```

```
class fclass {
  float f(int num) {
    temp : float := 9/5*num;
    return temp;
  }
}
```

Java does not support the notion of dynamic class membership, so if the `attach` and `detach` functions are to be implemented, they have to be supported by an additional layer implementing the dynamic mechanism. However, it is not necessary to implement them, because they are designed to model the dynamic movement of mobile entities sharing some common migration patterns. In reality, this situation can be represented using different ways. For example, in applications with code mobility, it can simply realised by a mechanism that is capable of sending multiple pieces of code to different locations concurrently.

4. *Bags are implemented as user-defined Java classes.* A bag representing a data object is implemented as an instance of an ordinary Java class. If the definition includes an initialisation assignment, an additional statement is required, as well as a constructor to enable object creation using the given values.

```
b1 : bag {
  name : string;
  age  : int;
} := ("John Doe",30);
```

```
class P {
  String name;
  int age;
  P(String n, int a) {
    name = n; age = a;
  }
}
P b1 =
  new P("John Doe",30);
```

The implementaion of a bag which represents a vehicle will be discussed in Section 5.4.5.

5. *Suitable container classes must be provided and organised as necessary.* As mentioned in the beginning of this section, all mapping definitions must be placed in an appropriate position to conform to Java's syntax. Methods and object instantiations must be wrapped into classes. Additional class definitions can either go into the same wrapper class or be promoted as top-level classes.

These rules provide guidance for writing and organising implementation classes. Class organisation is crucial as it affects how framework classes are incorporated in a program.

Composition of mapping constructs becomes the responsibility of programmers. They need to *weave* the mobility components and their functionality counterparts to form a complete program. At this stage the high-level abstraction carried by the mobility components melts and disappears, mixing with that of the functionality components. There is no more distinction between mobility and functionality components, and programming is subject to object-oriented practices.

5.4 The Design and Implementation of the Framework

The crucial aspect in designing the framework is to preserve as much as possible the conceptual abstractions of Mocha constructs. They are needed in order to provide the overall view of the architecture of an application. The solution is to keep the design of the framework classes as close to the original abstractions as possible. In particular, special attention must be given to the the context-based mechanism, because Java's procedural programming is not

compatible to the state-based model employed by the mechanism. The design of the framework's architecture is important to deal with this situation.

5.4.1 The Architecture of the Framework

The architecture of the framework supports the idea of making a vehicle as a central component in Java-based mobile applications. Its components highlight the mobility of functional components through context-awareness, as shown in Figure 5.1.

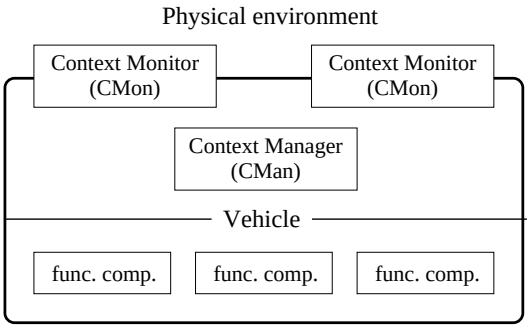

Figure 5.1: The architecture of the framework

A vehicle's structure is divided into two parts. The main part exhibits the vehicle's role as a container for functional components. The other part contains components for context management which represents the vehicle's mobile capability.

The container part can be implemented using normal class definition. The reference of any functional component that needs to be relocated is placed here, making the functional classes part of the vehicle class. If the vehicle object moves, any functional object that is defined within its class will also move.

The context management part distinguished a vehicle class from other classes. It basically consists of two elements: **context monitors** (CMon) and the **context manager** (CMan). A context monitor acts as a *sensor*

for the vehicle. It is created for each context listened by the vehicle. It detects any context state change and reports it to the context manager. The context manager is the core of the context processing. It does context information preprocessing if necessary, and based on the result, performs context evaluation and triggers action execution.

5.4.2 The Event Handling Mechanism

Recall that the operation of Mocha's mobility mechanism makes use of the concept of event (see Section 3.4.2). An event for a particular context state is generated when the context state evaluates to true, and the event persists as long as the state holds.

Figure 3.3 shows that Mocha events are continuous. However, it is not possible to represent the continuous state of an event in a discrete programming environment. Instead, the framework captures the state transitions that mark the beginning and the end of an event. To achieve this, it uses a different notion of events. An event in the programming framework (as opposed to a Mocha event) is used to signal a state change that toggles a Mocha event. Such an event is represented by an object that carries a value representing the context value associated with a Mocha event.

```
public class Event {
  Object value;
  public Event(Object v) { value = v; }
}
```

An event object is used in an *event-handling* mechanism, like the one adopted by Java's graphics programming environment [AG97]. The mechanism involves two entities to whom an event is associated. The first is an *event source*, and the other is an *event listener*. An event source is an object where an event object comes from. An event listener is the place where an event object goes to.

In the framework, the event source role is played by a context monitor. Executing on behalf of the environmental element it listens to, it triggers an event by creating an event object every time it detects a context state change during a measurement or when notified by an external source. The event object is then dispatched to all subscribed listeners.

The event listener role is played by a context manager. It listens to events generated by each context monitor. To do this it must subscribe itself to the context monitors. The scheme is shown in Figure 5.2.

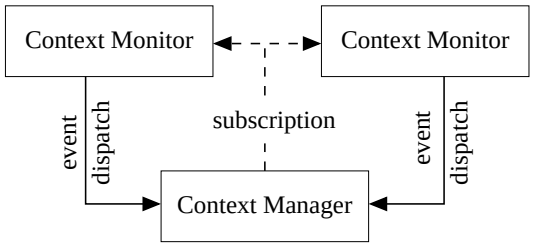

Figure 5.2: The relationship between context monitors and a context manager

5.4.3 The Design of a Context Monitor

A context monitor is actually not a part of a vehicle. It is designed to represent a context which is completely distinct from a vehicle. However, at the implementation level, it could become impractical to separate the two entities, since managing the interaction of a vehicle and a context object in a distributed computing environment is not a trivial task. For example, a context object should be available at any location a vehicle is visiting. This can be realised either by object replication or central management, but neither of these methods are easy to program.

Instead of placing a context monitor at the environment side, the framework chooses to incorporate a context monitor to a vehicle. However, a context monitor still acts on behalf of the context it represents. It lies in the boundary between a vehicle and the physical environment, and can

be considered as a *peripheral* part of the vehicle. Monitoring is performed by observing a change of an object's value representing the current context value.

A context monitor can work in the *active* or *passive* mode. In the active mode, it actively performs a monitoring activity, for instance by doing measurement on some physical quantities (e.g., network speed or hard disk space availability) at certain times. The returned value becomes the current context value, and if it is different from the previous value, a context state change has occured.

In the passive mode, a context monitor passively waits for other system components to *notify* a change. This mode is suitable in situations where the context monitor relies on an external source in obtaining information about a context state change. It happens, for example, in mobile agent applications, where the run-time system that implements a location can notify a context monitor of a mobile agent if it arrives at a new location.

The design of the template class that represents a context monitor is shown as follows.

```
import javax.swing.event.*;
import java.util.*;

public abstract class CMon {
  Set cvals;                    // context values
  EventListenerList listeners;  // event listeners

  public abstract boolean checkValue(Object v);
  public void addEventListener(EventListener l);
  public void removeEventListener(EventListener l);
  void dispatchEvent(Object value);

  public abstract Object monitor(Object arg);
  public Object accept(Visitor v, Object arg);
}
```

Recall in Section 3.4.1 that a context is represented by a set of context values. The `CMon` class contains a reference to a set object representing a

domain of the context it represents. The set object is useful if the context domain contains a finite number of context values. The class provides an abstract method, `checkValue`, for checking whether a value belongs to a set or not. The method is also useful for context with infinite number of context values. Instead of storing the context values in the set object, a class monitor can directly use a value if it belongs to the context.

The `listeners` object in a context monitor holds a list of event listener objects. Addition and removal of event listeners are done using the `addEventListener` and `removeEventListener` methods, respectively. The methods accept an event listener object, which is a context manager in this case.

Event dispatching is performed by the `dispatchEvent` method. It creates an event object containing a specific value by calling the `createEvent` method and sends it to all members of the listener list.

The event dispatching method is called when a context state change occurs. The notification can come from the context monitor itself, in this case if the context monitor actively detects for such a change, as indicated by an invocation of the `monitor` method. Instead, if a context state change is informed by an external source, the `accept` method is called. In return, this method will call the `visit` method of the caller, before resuming with the event dispatching. The reciprocal behaviour implements the *visitor* pattern [GHJV95], which allows non-intrusive modification of the object behaviour.

5.4.4 The Design of a Context Manager

A context manager has the responsibility of maintaining subscriptions to one or more contexts, listening to a context state change event, and performing mobility state evaluation. Its class design that reflects these tasks is shown as follows.

```
import java.util.*;

class Subscription {
  CMon cmon;          // context monitor subscribed
  Object current;     // register for current context value
  Vector previous;    // register for previous context values
  void update(Object newvalue);    // update the registers
}

public abstract class CMan implements
  EventListener, Visitor {
  // acts as an event listener and a visitor

  Vector subscr;    // contains list of subscriptions

  public void subscribe(CMon evsrc);
  public synchronized void acceptEvent(Event event);
  public boolean updateRegisters(Event event);
  public abstract Object assignSemantics(Object cval);

  boolean before(Object cval, int count);
  boolean at(Object cval, int count);
  boolean after(Object cval, int count);
  public abstract void eval(Event event);
}
```

The `subscr` object holds the list of context subscriptions. For each subscribed context, a context manager instantiates an object of the class `Subscription` to hold context monitor, the current context value, and a list of past context values. These values provide the information required to determine the mobility state of the vehicle. Manipulation of context values is performed by calling the `updateRegisters` method. Figure 5.3 shows the structure of a context manager with its subscription objects.

As an event listener, a context manager object has to implement the `EventListener` interface. An event object is passed to a context manager by calling its `acceptEvent` method. This method is a gateway to a series of processing that initiate a mobility state evaluation, which makes use of the `before`, `at`, and `after` methods (this is explained in Section 5.4.6). If a raw context value is passed along with an event object, it can be assigned a more intuitive meaning using the `assignSemantics` method.

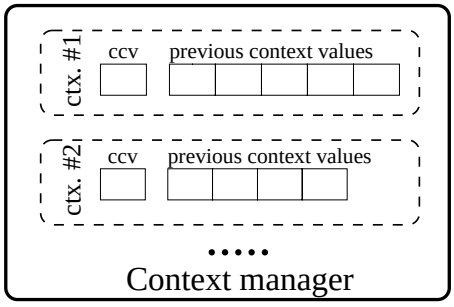

Figure 5.3: The structure of a context manager

5.4.5 Implementing a Vehicle

A vehicle is distinguished by its context-awareness. Since this functionality is provided by a context manager, it perfectly makes sense to subclass the `CMan` class to represent a vehicle. A subclass basically covers two things: realisation of the parent's abstract methods, and inclusion of classes and methods that implement the functional components. The inheritance is shown by the following code portion.

```
public class aVehicle extends CMan {
  public Object assignSemantics(Object cval) { ... }
  public void eval(Event event) { ... }
  ...
}
```

In a Mocha specification, a vehicle is a *passive* part of a bigger but transparent control system. A vehicle is subject a control mechanism external to itself when it is involved in a context statement. In the scope of a context statement, the execution control applied to all vehicles lies in the top-level program. This configuration allows mobility control of the overall system to be applied centrally.

The centralised approach requires a top-level application object to hold all vehicle objects and implementation of context statements which control the mobility of the vehicles. However, programing this approach can get

very complicated when routines from a mobility programming toolkit (e.g., μCode [Pic98] or Voyager [Obj99]) are embedded in vehicle objects as a relocation engine to enable code migration. The central role of the top-level application object requires it to be accessible by all vehicles, and an intuitive solution would be making it fixed to a particular location. Program complexity increases as vehicles located in different locations have to be controlled by a centralised object. Non-trivial processings are split between objects located in different places, and they have to be centrally coordinated for the application to run properly.

Alternatively, a vehicle can be designed as a self-contained entity. In the self-controlled approach, the controller role is taken by an individual vehicle. A vehicle proactively control its own mobility without the presence of any central controling mechanism. Instead it uses a generic context-based mechanism which may utilise a relocation engine as a primitive to enable code migration. This approach is a simplified implementation of the Mocha model, but it removes much of the programming complexity in implementing the centralised control mechanism.

A self-controlled approach requires a vehicle to be self-sufficient, in the sense that it should be able to acquire all information required to control its mobility, and perform mobility state evaluations to achieve this purpose. The first requirement is satisfied by the collection of context monitors. The second requirement is met by the context manager through the `eval` method. The method is designed to implement a context statement. It basically contains a series of `if` statements for mobility state evaluation.

5.4.6 Mobility State Evaluation

The main part of the `eval` method implemented in a vehicle object is a collection of `if` statements for mobility state evaluation. Implementing an

evaluation expression is straightforward since such an expression is simply a boolean term. The `before`, `at`, and `after` methods have to be realised as well for this purpose.

A mobility state evaluation can be performed using simple algorithms working on information provided by context value registers. The algorithms assume a given pair of arguments, *value* and *count* which represents a context value and its n^{th} occurrence, respectively. They also need $num(value)$ that denotes the number of occurrences of a given context value so far. The algorithms are given in the following.

- The *before* state:

```
calculate num(value);
if (num(value) < count)
  return true;
else
  return false;
```

- The *at* state:

```
if (current_value == value)
  return true;
else
  return false;
```

- The *after* state:

```
calculate num(value);
if (num(value) > count)
  return true;
else
  return false;
```

The crucial part of mobility state evaluation is the execution timing, i.e., when an evaluation should take place. The Mocha model specifies that evaluation should happen if there is a context state change. In the programming framework, it means an evaluation is started only when a context

monitor sends an event to the context manager. In this scheme, the invocation of the `eval` method actually depends on the *monitoring policy* adopted by all involved context monitors. A context monitor can do monitoring either actively or passively, regularly or on-demand. In all cases, monitoring thread is usually asynchronous to the thread running the vehicle. To achieve this, a context monitor should run on a different thread from the main thread. Its operation can be started when the thread is initialised (i.e., when the method `run` is called). This is shown in the following code.

```
public class NetworkMonitor extends CMon
  implements Runnable {
  ...

  public Object monitor(Object arg) {
    // perform network monitoring
  }

  public void run() {
    monitor(null);
  }
}
```

A typical vehicle class definition that shows how to subscribe to a context and start a monitoring thread is shown in the following code. The code shows that the context monitor runs on a different thread from the main program's thread.

```
public class MyVehicle extends CMan {
  // definition of all abstract methods goes here

  public void eval(Event event) {
    // specify mobility state expressions here
  }

  public static void main(String argv[]) {
    NetworkMonitor mon = new NetworkMonitor("mon_1");
    MyVehicle vhc = new MyVehicle();

    vhc.subscribe(mon);          // subscribe to 'mon'
    new Thread(mon).start();     // activate the monitor
  }
}
```

The execution flow of a mobility state evaluation is illustrated in Figure 5.4. It shows four separate monitoring threads operating on the same context manager. Whenever a context monitor detects a context state change, it creates and dispatches an event which will be captured by the context manager through the `acceptEvent` method.

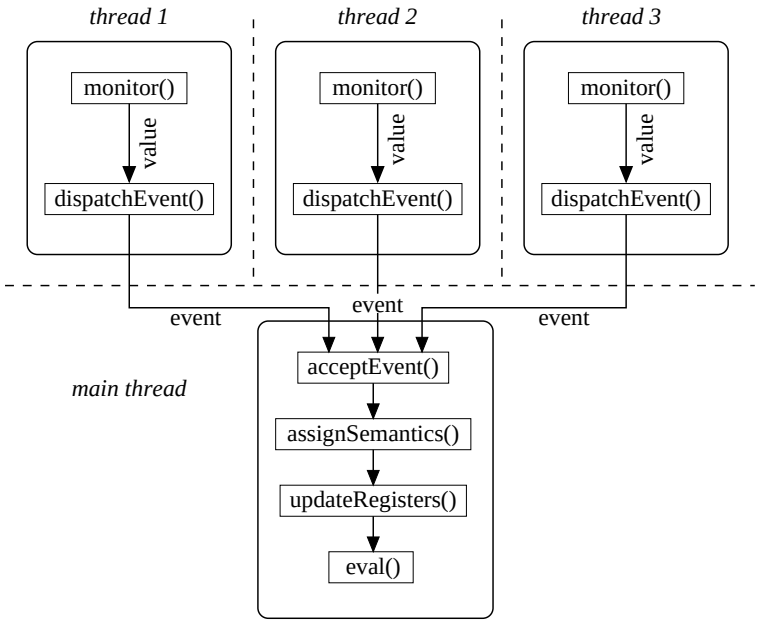

Figure 5.4: The flow of context value processing

The first process is an optional semantics assignment of a context value. The context value passed on to the `acceptEvent` may come directly from a physical measurement and may need to be given a meaningful attribute before it can be used further. This can be done by implementing the abstract method `assignSemantics`. An example of using this method is to translate a network speed reading to a human-oriented categorisation such as "low" or "high".

The next processing step is to update the status of the context value registers. The current context value register is replaced by the value carried by the event object and pushed into the list of previous values.

Finally, mobility state evaluation is performed by the `eval` method. Given arguments of the context value being processed and a counter that indicates the n^{th} occurrence of the corresponding context state, the *before*, *at*, and *after* state can be determined using the algorithms presented in the beginning of this section.

As shown in Figure 5.4, the series of processes are executed in a separate thread from those of context monitors, and must be synchronised to protect the integrity of the results. In this processing mechanism, execution of the methods depends on a monitoring thread, passively waits for an event notification. At the same time, multiple threads are allowed to participate in the mechanism. The net effect is an execution semantics similar to that of the execution model of context statements.

5.4.7 Adding a Relocation Engine

The purpose of a relocation engine is to enable a program component to perform some form of real movement. An example of a relocation engine is the mechanism that allows a mobile agent to move from one machine to another. A relocation engine is commonly provided by programming toolkits that have a feature to send code to remote locations, which is normally based on remote communication techniques such as Remote Method Invocation (RMI) [Mic98] or network socket programming.

The framework uses the code migration features of the μCode toolkit [Pic98]. Unlike other toolkits, μCode is lightweight in the sense that it emphasises on fine-grained code mobility. Code relocation is performed at the smallest unit of mobility (i.e., classes and objects). Additional semantics (e.g., autonomy in mobile agents) is not part of the mechanism but can be added on top of it. This provides a flexible mobility mechanism that allows various mobility abstractions to be built on top of simpler constructs.

μCode's modular design makes the embedding of a relocation engine in the framework not difficult. What is required is to define the unit of mobility, then embed the engine to that unit. Since the unit of mobility is a vehicle, and a vehicle is implemented as a subclass of a control manager, it makes sense to implant the engine definition to the `CMan` class.

The engine should work transparently in providing mobility feature to the framework. Its operation should be hidden from programmers. In μCode, any code relocation can be viewed as a change of execution environments. This fits perfectly with the context-based model. μCode's migration operation can therefore be viewed as a transparent functionality that allows code to change locations. In the framework, the functionality can be integrated using the same event-handling mechanism.

An intuitive way to do the integration is to build a location abstraction on top of μCode's location server, called `MuServer`, and makes such an object capable of emitting an event every time a vehicle is coming to that location. A vehicle can subscribe to this event source, and be notified when it enters the scope of the location. The problem with this approach is that a vehicle has to subscribe to every available location server. This becomes impractical if there are many of them.

The framework takes a less elegant but simpler solution. It relies on the autonomy of a vehicle. Instead of expecting a location server's notification, a vehicle notifies itself when it arrives at a new location. This is done by generating an event, which is sent to the vehicle's own context monitor. The context monitor will capture the event and process it as if it came from an external source. The mechanism is shown in Figure 5.5. It eliminates the need for an explicit location server. It is important to keep the *self-notification* mechanism transparent. Programmers only need to know the destination locations, while the details are taken care of by the framework.

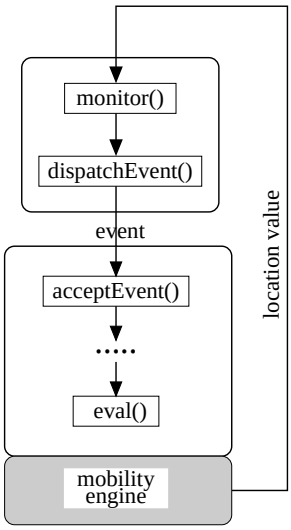

Figure 5.5: Context value processing with the presence of a relocation engine

This approach can be implemented using μCode's `MuAgent` mobility abstraction. The main characteristic of this abstraction is its autonomy. A `MuAgent` object can be programmed to visit certain locations and do some actions without requiring external intervention. The following code shows how a relocation engine is embedded in a context manager class definition.

```
import mucode.abstractions.*;

public abstract class CMan extends MuAgent
  implements EventListener, Visitor {
  ...
}

public class MyVehicle extends CMan
  implements EventSource {
  ...
}
```

`MuAgent` object's autonomy is implemented by a thread object. Java has a limitation is performing thread migration: it cannot save the execution state during a migration, so a thread has to be restarted when it arrives at the destination. This limitation creates a restrictive effect on Java-based programming toolkits, as it becomes impossible to have *strong mobility* that

demands execution state to be migrated along with a thread. The framework, on the other hand, is not affected by the limitation. The separation of mobility and functionality with the reactive execution model insists that execution state (i.e., context state in this case) to be evaluated every time a state change occur. This eliminates the need for saving such a state during code migrations.

The framework can make use of the reactivation of a thread to initialise context monitoring at the new location. This is done by specifying it in the `run`[1] method of a vehicle, which is the execution entry when the thread is restarted. The itinerary of the vehicle can be implemented using a sequence of destinations. The next destination is determined by the first element of the sequence, which is removed from the sequence before the migration takes place.

```
public class MyVehicle extends CMan
  implements EventSource {

  String[] dest = {"host1","host2","host3"};
  Seq dests = new Seq(dest);
  ...
  public void run() {
    // start the context monitor threads
    network_monitor.t.start();
    location_monitor.t.start();
    // wait for the monitoring threads to finish
    // before performing a migration
    network_monitor.t.join();
    location_monitor.t.join();
    // remove the head of the sequence
    String next = (String) dests.remove(0);
    // migrate to the next destination
    if (dests.size() > 0) go(next);
  }
}
```

It should be noted that a vehicle whose mobility is driven by an engine requires a terminating context monitoring threads (i.e., the threads do not

[1]A thread object has to define a `run` method. The method becomes the entry of execution when the thread is started.

run forever). They have to be stopped, serialised, and sent to a remote machine during a migration, then woken up again in the new location.

5.5 Discussion

The previous section discusses how reactive mobile systems can be implemented in Java. This section describes the lessons learned from the framework implementation. It analyses the applicability of Mocha's modeling paradigm in the implementation stage of a software development project. The analysis focuses on the fitness of an existing programming language (i.e., Java) in implementing the modeling paradigm. The purpose of this section is to identify programming-specific requirements for a natural implementation of the paradigm.

5.5.1 General Issues

There are substantial differences between application implementation following Mocha approach and that following the conventional approach. The conventional approach treats a mobile application the same as other types of applications. Its implementation is based on a model that cannot capture its unique aspects. Such a model fails to provide a strong basis for supporting mobility. Consequently, its implementation inherits this weakness, and is often able to provide partial mobility support.

On the other hand, Mocha offers a more conceptual approach. Its modeling framework provides a uniform model for mobility to handle different types of requirements, and at the same time allows its modeling abstrations to be gradually refined and implemented. Since programmers have a unified view of mobility, any mobile application could be implemented using the same development framework.

The differences between the two approaches are illustrated by a mobile agent application. In the traditional approach, a mobile agent is modeled as a collection of functionalities that can be sent to a remote machine. As shown by many agent-based programming toolkits (e.g., Voyager [Obj99], Aglet [LO98], Sumatra [ARS97], and Agent Tcl [GKCR97]), the concept of migration is as a functionality. Consequently, this mobility feature is implemented using the programming framework designed for implementing functional abstractions (e.g., the `moveto` method to perform a migration).

Problems arise when the software requirements expand, for example, to include support for context-awareness as well. The conventional approach treats code mobility differently from the new requirements (e.g., code migration vs asynchronous programming), therefore their implementation attracts different programming approaches, increasing the complexity of the program.

With Mocha, a mobile application development starts from an abstract model that views mobility as a common feature of any entity with a potential to move. More importantly, mobility is treated separately from the functional aspects, and handled by a universal mechanism. In the agent example, agent migration is handled separately from its task execution. When the requirements expand, the same mechanism is used to deal with user migration (e.g., to perform context awareness). The implementation does not have to use different approaches, therefore reducing program complexity.

The genericity of Mocha allows redefinition or extension of the requirements of a mobile application to be implemented in an elegant manner. This not only covers those related to implementation, but those related to software allocation as well. For example, a user requires that task execution of the mobile agent is now carried out by physically moving the computer to the destinaton (i.e., by the user). The policy change requires radical modification of the software requirements. Using Mocha's approach, such a modification is simplified by isolating the mobility aspects from the functional aspects

(i.e., the design and implementation of functional modules does not have to be modified).

5.5.2 On Programming Paradigm

Object-orientation is a programming paradigm that is designed with the concept of modularity in mind. The class construct plays a vital role in encapsulating units of abstraction. Clean boundaries between design abstractions enable modular implementation of program components. With this feature, it is easy to implement Mocha constructs into Java objects. It is straightforward to transform most Mocha object definitions into equivalent Java classes and object instantiations. Mapping of Mocha functions requires a bit of twiddling, as writing them as methods needs user-defined classes that wrap the methods.

Object-orientation also performs very well as an integration tool for mobility and functionality components. Once mobility components have been transformed into Java objects, they can be seamlessly composed with functional components that may have been developed before. It should be noted, however, that from an implementation point of view, similar well-composed programs may be produced from other programming paradigms (e.g., procedural). This is because at the programming level, all concepts have been uniformly translated using the same programming framework. There is no more difference in abstraction levels of components from different domains, like those found in Mocha. In this situation, the selection of object-orientation may not have preference over other programming paradigms. For example, an implementation in C language would probably give a similar quality in terms of program composition.

In summary, from Mocha's point of view, the selection of a programming paradigm for implementation is not a crucial issue. However, it becomes

important during an actual implementation (i.e., programming) activity, as such work involves external aspects that are outside the realm of Mocha model (e.g., addition a mobility engine, need of additional toolkits, and possible program extension or modification). When new components must be integrated, program composition becomes an important issue, and the selection of a capable programming paradigm becomes relevant at this point.

5.5.3 Semantics Implementation

It is expected that the programming environment used in an implementation should be able to preserve the semantics of the specification language while realising it in an actual execution environment. In developing the framework for Mocha, a big implementation issue is the incompatibility between the sequential nature of Java program execution and Mocha's reactivity. This incompatibility becomes an obstacle in the realisation of context statements. A context statement construct requires a non-terminating loop with external, possibly asynchronous inputs that drive loop iterations.

The framework circumvents the problem by moving the loop mechanism away from a vehicle as the main controller part of a program. The responsibility of performing the loop is taken over by context monitors which run on separate threads. The vehicle, running on the main thread, passively waits for an event indicating a context state change. This solution, however, has disadvantages. First, there is a semantic change to context statements as the notion of loop disappears. Furthermore, it requires non-transparent thread programming. This adds some burden that can distract programmers from their main tasks.

The disadvantages show that Java is not a natural language for implementing Mocha's execution model. A better solution could be implemented using a linguistic mechanism that naturally captures the reactive semantics

of a context statement. Moreover, the mechanism must not *impurify* the semantics of the underlying programming paradigm, because it can reduce the effectiveness of the paradigm, particularly when the language is also used for programming the functional aspects.

One possible way to implement the requirements is to build the linguistic mechanism on top of the original language. In Java environment, this approach has been implemented, for example, by Junior [HSB99] and its predecessor Sugar Cubes [BS98]. Both systems add reactivity to the original Java execution mechanism. Reactivity is achieved through new constructs called *reactive instructions*, which are executed using logical execution engines called *reactive machines*. The set of reactive instructions provide necessary operations in a reactive system. Execution engines encapsulate the execution behaviour of any operation performed on it, transforming the sequential execution model of Java to the one that exhibit reactivity and parallelism. The idea of using reactive machines to facilitate the new execution behaviour is interesting, as it isolates the new behaviour from the original one. This minimises any incompatibility that may occur due to the addition of the new mechanism.

5.5.4 Multiple Mobility and Rendevous

The framework for Mocha does not implement multiple mobility with rendezvous arrangements. If multiple mobility is to be implemented, a separate thread has to be created for each context statement defined in a model specification. This thread has to cooperate with other threads representing vehicles and context monitors. Moreover, this thread cooperation has to work in a distributed environment, as vehicle and context monitor threads may migrate to other locations.

The difficulty in implementing the scheme is due to lack of high-level support for multithreaded programming and synchronisation over distributed objects. Java's thread management is based on the concept of *monitors* [Hoa74]. The low-level nature of the concept makes implementation of the scheme complicated. Moreover, with the current design of the framework, such thread management cannot be made transparent as it occurs in a user-defined part of the developed program.

One way to approach this problem is to built a high-level, specialised layer on top of the original Java mechanism. The layer abstracts the mechanism required for multithreaded programming in a distributed environment. Perhaps the biggest challenge in implementing the layer is to integrate the thread migration mechanism that is performed by a programming toolkit (e.g., μCode in the case of the research experiment).

To summarise this chapter, the experience of developing the programming framework for Mocha has proved that the context-based approach is applicable in the traditional application development course, in which implementation is carried out in a well-supported programming environment. Although Java is not an ideal language for Mocha, it can still be used as an implementation language with some limitations.

It should be noted that mobility support can be developed using implementation environments other than a programming environment. This will be discussed in the next chapter. The research described in the next chapter also explores the implementation aspect of Mocha, but it takes a different course. Instead of working with a programming environment, it develops mobility support in an operating system shell environment.

Chapter 6

An Operating System Environment for Mocha

6.1 Introduction

Chapter 5 discussed the development of a programming framework for Mocha to allow the implementation of a specification in a a conventional programming environment based on Java programming language. This chapter discusses the implementation aspect of Mocha as well, but it pursues a different direction. It focuses on implementation on an operating system environment. The research described in this chapter develops a user tool with mobility support capability which runs on a Linux shell environment.

The purpose of the work described in this chapter is to complement the work described in Chapter 5. Together, they aim to demonstrate the applicability of Mocha's approach to support mobile application development with different requirements, possibly on different computing environments. In Chapter 5, the emphasis was on Mocha's usability in a programming environment where application development is well supported by a programming language. In this chapter, the same usability criteria is exercised in an environment where a program is constructed by configuring other programs as building blocks.

The implementation course described in this chapter develops a prototype of a Linux tool that provides *spatial sense* to mobile users. Linux already provides a platform for working in a distributed computing environment. However, a mobile user needs to be aware of his or her location to take advantage of his or her mobility, and this feature is not supported by Linux. Spatial sense gives a user the feeling of location awareness. It allows the user to detect his or her current location, to track past locations, and to plan visits in the future. Moreover, the awareness can be used as a hook to execute actions using location-based reasoning.

The practical usage of the tool is promising. It extends the capability of Linux, and Unix in general. It shows its usability when a user wants to organise his or her activities based on the locations visited. It also allows planned action execution on remote machines, which is also expressable using the location-based framework.

The following sections describe the prototype in detail. It starts with a review of the location context and how mobile users deal with this in a Linux environment. The weaknesses of current techniques are discussed, followed by an explanation on the proposed approach to improve the situation. The main part of this work, an extension to a user environment, is then discussed. Explanation on some implementation issues and possible enhancements conclude this chapter.

6.2 The Location Context

Linux is well known as an operating system with out-of-the-box networking features. When installed on a computer, it does not require additional software tools to connect to a network. Linux also comes with a suite of networking tools that make life easier for both normal users and network administrators.

The capability of being connected to a network is essential for a mobile user. This feature allows access from anywhere, masking out the location distribution that arises from the mobility of the user. The user can execute his or her processes on a remote machine from any computer that can reach the remote machine.

In mobile computing, locations are normally made visible to applications so that a computation can take advantage of their presence for reasoning purposes. This requirement can be implemented on top of the distributed computing techniques. In particular, a visit to a location can be modeled by the state of being connected to a network or a remote computer. In a mobile environment, the presence of a user at a particular location is shown by his or her network connection, at any time. In this model, a location can be thought as an abstract spatial entity, bounded by the network or host to which the user is connected.

There are two perspectives on locations, as shown in Figure 6.1. The first is the high-level, user-oriented perspective, shown by the left side of the picture. It views a user having a high-level connection (i.e., a remote session) to a remote host. The main characteristic is that there is no conceptual distance between the user and the remote computer. It creates a conceptual image that the user is *locally* using the computer. Remote connections using **telnet** or **ssh** are examples of connections that create this perspective.

The low-level, connection-oriented perspective, on the other hand, abstracts the physical link between the two computers (shown by the right side of Figure 6.1). It simply states that there is a communication link between them so that higher level form of communication can be built on top of it. This is a building block for the high-level perspective. A low-level connection constructs an image that the connected machine becomes local to the other machine. If the latter is a part of a bigger network, the connection states that the connected machine also becomes part of the network. Examples are

shown by network connection using ethernet or PPP (point-to-point) protocol. With this perspective, it is possible to build the abstraction of network applications (i.e., applications that run on a network).

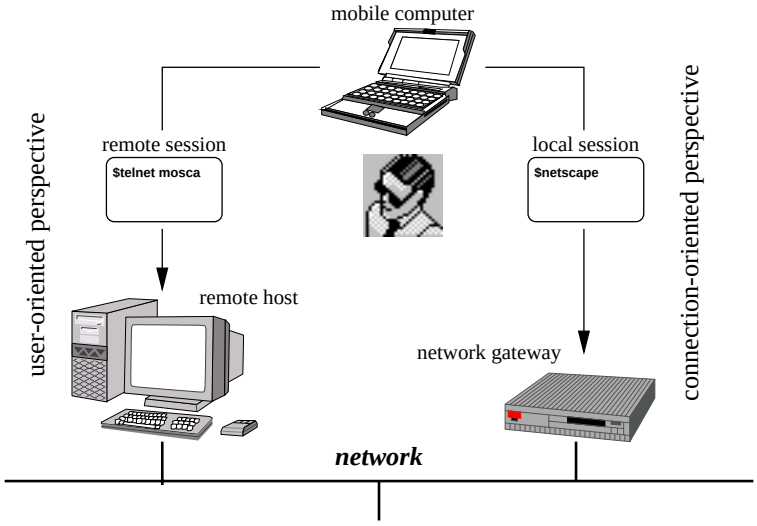

Figure 6.1: Perspectives on network connections

Using the connection-based perspective, the mobility of a user can be expressed as connections that change dynamically over time. However, this argument is not symmetric. The reverse statement that the dynamics always represents mobility is not true. This is because there is no association between a location *representation* (i.e., a machine with an IP address) and the *geographical* location of a user. For example, a series of **telnet** sessions to different machines do not necessarily show physical movements, as they might be done from the same machine at the same physical location. Conversely, virtual private networks allow wide-area networks use the same network class, masking connections made from different geographical locations. This asymmetry is due to the fact that the underlying distributed environment transparently hides the geographical aspect of location. It is actually up to the user to assign a semantics to a series of connection changes: whether it *actually* represents the user's physical movements, or it is just a series of connections with no semantics of mobility.

A location to which a user is connected provides a locality for the user's computation that is executed there. It provides the required computing source and governs the execution to follow its rules and restrictions. This characteristic fits the description of a context (see Section 1.4), therefore it is convenient to promote user location as a context. This allows Mocha's paradigm to be applied to developing mobility support in a Linux environment.

A potential application for having the location context in a Linux environment is to control task execution based on the user's location. The principle is similar to the **cron** program that schedules tasks based on time. The difference is that now the tasks are scheduled based on the spatial property. From here the possibility can be expanded. For example, a user can specify the tasks that need to be executed in the location he or she is visiting. Task execution can be automated when a connection is made to the host representing the location. Even a user migration can be automated by arranging a sequence of connections. Moreover, in an environment where services are charged based on their usage [LSP+97], accounting can be performed on a location basis.

Location awareness cannot be directly implemented in Linux since traditionally Linux does not support the model. Linux provides some primitive support though, for instance through the **ipconfig** command that shows information about a low-level connection. Information on a high-level connection (e.g., a **telnet** session), on the other hand, can be found by examining either the contents of the **/proc/net/tcp** file which reflects the real-time image of the kernel's data structure for a TCP connection, or the output of a **ps -ax** command. Given the above primitives, the purpose of this implementation project is to build a Linux tool that can make use of the location context. The proposed design is discussed in the next section.

6.3 LaseriX: Location Awareness for a Linux User

This section describes **LaseriX**, which stands for **l**ocation-b**a**sed context for u**ser**s of L**inux**.

LaseriX is a shell program, which runs on the user space (i.e., instead of the kernel space) (see [SG99], chapter 22 for a discussion on how Linux manages running processes). When LaseriX is active, it can monitor the current location of the user and keep the record of past locations. The semantics of a user's location is defined by the type of connection he or she is making. With the user-oriented perspective, LaseriX explicitly refers to a particular machine in a network where the user is having a session. With the connection-oriented perspective, LaseriX simply states that the user's computer is connected to a network or it works in a stand-alone mode.

The concept of context state can be applied to either semantics of user location. A context-state determines whether the user is within the location (or is being connected to a network) or not. The context value changes dynamically as the user travels to different locations. Once a context state system is established, a mobility state system can be laid on top of it. It describes the mobility of a user viewed from a temporal perspective.

LaseriX allows a user command to be bound to a location referred to be either a user-oriented or low-level connection. Binding to the user-oriented connection makes the execution to be performed on the remote machine. Binding to the lower level connection makes the command be executed locally on the user's machine. The semantics of a binding follows the action execution model of Mocha, as described in Section 3.4.7. When a particular mobility state becomes true, the commands are executed. The location provides execution environment for the commands.

6.3.1 Architectural Design

LaseriX is implemented as a collection of programs that provides location awareness to a user's environment. It has two parts: a connection monitor and the task scheduler. The purpose of a connection monitor is to constantly monitor user location changes. It is activated by the task scheduler, which is designed to execute specific tasks based on the past, present, and future locations of a user. It consults a set of execution rules, and based on the input provided by the connection monitor, it triggers the execution of the user commands. The architecture of LaseriX is shown in Figure 6.2.

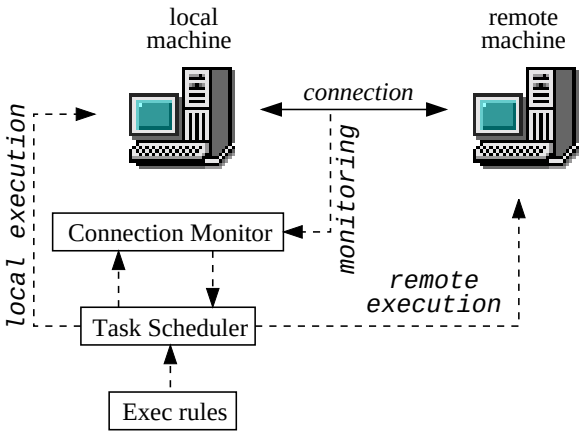

Figure 6.2: LaseriX components

Figure 6.2 shows a connection between a local (possibly mobile) and a remote machine. It can either be a user-oriented or a low-level connection. The local machine has LaseriX installed while the remote machine is just a normal Linux installation. The connection monitor periodically monitors the status of the connection. More specifically, it records any instantiation or termination of an ethernet or point-to-point link, and remote sessions using telnet or secure shells (ssh). This information is then logged to a file which can be read by the task scheduler. It will consult a set of execution rules to launch some specified commands, either locally or remotely. The details of each component are discussed in the following sections.

6.3.2 The Connection Monitor

The purpose of a connection monitor is to monitor context state changes recording the events generated when a user is entering and leaving a location. The connection monitor works by examining information provided by the Linux kernel. It expects the IP address of the remote machine. Determination this address in a low-level connection is straightforward. The **ifconfig** tool provides the necessary information about all network interfaces of a computer. The required IP address of the remote host can be obtained through these interfaces.

To get the same information in a user-oriented connection, LaseriX uses a simple approach. It just reads the output of the **ps ax** command. The information is available since when a user makes a connection using **ssh** or **telnet** command, the destination host is specified as an argument. However, this method does not work with an interactive telnet mode.

As the location context is represented by user connections, changes in context state are translated into the establishment and termination of connections. Therefore a connection monitor needs to record both events. This is done through a log file created at the beginning of the monitoring process. The log file contains entries with the following format.

⟨PID⟩ ⟨target_host⟩ ⟨timestamp⟩ ⟨status⟩

For a user-oriented connection, the ⟨PID⟩ column shows the process identifier of the executed connection. For a low-level connection, it contains the network interface of the connection. The ⟨target_host⟩ is the remote machine for a high-level connection. In a low-level connection, this refers to the network address of the machine (for **eth*** interfaces) or the address of the other connection point (for **ppp*** interfaces). The ⟨timestamp⟩ is a time stamp for the connection, and finally the ⟨status⟩ column shows whether the connection is active or it has been closed.

A connection monitor invocation launches a **ps ax** and an **ifconfig** commands and extracts relevant information. When it finds one, it is compared to the entries in the log file. If it is not there, then it represents a new connection. Its entry is created, given a time stamp and an active status, and appended to the log file. Similarly, if an entry in the log file is not present in the current process status, it is marked as terminated. Each connection has its own entry, and an entry will never be deleted from the log file even though the connection has been terminated.

6.3.3 Execution Rules

To use LaseriX, a user must organise tasks that need to be executed during his or her travel into a set of execution rules and save them in a configuration file. An execution rule is an implementation of a context statement described in Section 4.4.5, and is used to control the execution of user commands.

An execution rule consists of two parts. The first part is a clause that represents a mobility state expression using IP addresses as context values. The operators `before`, `at`, and `after` are used to express mobility states with respect to connections made to the specified IP addresses. The second part is the sequence of commands that will be executed if evaluation of the expression clause yields a true value. The syntax of an execution rule is very similar to that of a context statement (shown in Section 4.4.5). Modifications are made to allow redirection of the command's output, as shown in the following.

execution_rule ::= ⟨mob_clause⟩ ' : ' ([**local**] ⟨command⟩ '; ') ∗

mob_clause ::= ⟨arg⟩ (⟨op⟩ ⟨arg⟩) ∗ ' > ' ⟨display⟩

arg ::= ['!'] (**before** | **at** | **after**) ⟨cntxt_val_expr⟩

op ::= '&&' | ' | | '

cntxt_val_expr ::= ⟨ipaddr_literal⟩ ', ' ⟨occur_expr⟩

occur_expr ::= ⟨digit⟩ | ⟨range_expr⟩ | ' * '

range_expr ::= '[' ⟨digit⟩ '..' ⟨digit⟩ ']'

display ::= ⟨string⟩ | **default**

The ⟨command⟩ tag represents a user command, which has to be separated by a semicolon from another command. Any Linux command can be specified in this part. The ⟨ipaddr_literal⟩ represents an IP address, either specified in numeric or string format. A location expression consists of three parts. The first part is the identifier of a location, denoted by its IP address. The second part is a numeric or range expression that denotes the number of visits to the location. The star ('*') symbol matches to any visit. The ⟨display⟩ symbol denotes where the output of the commands are sent. Regarding the negation symbol ('!'), the phrase `! at tarma,*` is interpreted as `(! at) tarma,*`, which states that it matches to any connection to any host *except* `tarma`. To say that the connection to `tarma` does not exist or is down, the rule must be modified as follows: `! (at tarma,*)`.

The explanation of how execution rules work is given through an example which uses the following specification written in Mocha language (refer to Section 4.4.5 for explanation about a context (`on`) statement).

```
LocationSet : set ipaddr := {mosca, tarma, insect, paragon,
                             mailhost, 130.194.224.46};

on user of LocationSet {
  user at mailhost/* : { pine; }
  user at paragon/1  : { cd work/thesis/;
                         latex thesis.tex;
                         dvips -o thesis.ps thesis.dvi; }
  user before tarma/5 : { ac -d; }
  user before mosca/3 &&
  user after mosca/1  : { echo "Users currently logged on:";
                          who | awk '{print $1}'; }
  user at 130.194.224.46 : { netscape; }
}
```

The rules that implement the specification is shown in the following. There is only one mobile entity in consideration, that is, the user itself. Therefore it is not necessary to express it explicitly as a vehicle.

```
at mailhost,* > /dev/pts/3:
   pine;

at paragon,1 > /dev/null:
   cd work/thesis/;
   latex thesis.tex;
   dvips -o thesis.ps thesis.dvi;

before tarma,5 > default:
   local ac -d;

before mosca,3 && after mosca,1 > default:
   echo "Users currently logged on:"
   who | awk '{print $1}';

at 130.194.224.46,* > default:
   netscape;
```

As mentioned earlier, a location provides the execution environment for the commands. Unless explicitly modified, any command defined in a clause is bound to the location specified, and is to be executed there. An implication of this situation is that it has to be possible to do a remote execution, i.e., to send a command from the local host to the host representing the location. This is not difficult as Linux provides some remote shells that allow commands to be executed remotely. For some security reasons, some remote shell commands such as **rsh** may not be allowed. It is the user's responsibility to have access to a more secure solution in this situation (e.g., by using **ssh** instead of **rsh**).

In the first rule of the example, the context value is a computer called `mailhost`. The rule states that every time the user establishes a **telnet** or **ssh** session to this host, the mail program **pine** on that host is invoked. The output is sent to the default terminal on the local host. LaseriX does not apply any ordering when trying to execute a command remotely. For

example, the **pine** interactive session may be executed in parallel to the user's remote session, so the output may have to be redirected to a different terminal (the **/dev/pts/3** in the example).

In the second rule, three commands are executed sequentially at the first session on the host `paragon`. Because no output is expected, it can be redirected to **/dev/null**.

The third rule illustrates the use of the `before` operator. The operation matches to any session to any host made before the fifth session on `tarma`. If it is desired to select only sessions on `tarma`, a range expression can be used: `tarma,[1..4]`. The `local` modifier tells that the `ac` command, which prints some accounting information, is to be executed locally. If the modifier is not specified, the command would be executed on every remote host visited between the specified session on `tarma`. It should be noted that the semantics of `before` requires a counter for the number of connections to be specified, otherwise it might cause some ambiguity if multiple connections to the same location are present. This also applies to the `after` operator.

The fourth rule consists of multiple mobility state expressions. The associated commands will be executed on every remote host that satisfies the criteria if the overall expression evaluates to true.

Finally the fifth rule denotes a low-level connection. An IP address indicating a modem server is given. This is to show that the local machine is connected in some way to a network associated with the modem server (e.g., using ethernet or point-to-point protocol). A low-level connection has the local host as the default for command execution. In the example, the web browser will be executed locally. Low-level connection is mainly used to organise execution of applications which require a computer to be connected to a network.

6.3.4 The Task Scheduler

The task scheduler is the main part of LaseriX. It performs a series of tasks: launching a connection monitor, evaluating execution rules against the current context state, and executing commands based on the result of the evaluation.

The task scheduler uses a loop that implements a polling mechanism. Each iteration performs a call to the connection monitor. If there is a state change, the scheduler will call the evaluation routine. The granularity of the polling time is important. A coarse granularity is preferred, although it may miss a short time connection, because setting it too frequent will waste computing resources since normally a connection needs to stay for a while to be usable (e.g., for Internet browsing, file transfers, or reading e-mail).

The operation of the task scheduler is governed by execution rules stored in a configuration file. The task scheduler reads this file and parses its contents to prepare the evaluation of the rules. Evaluation is driven by the polling loop and the result of the connection monitor (i.e., evaluation will only be performed if there is a state change). The following algorithms are used for evaluating the rules. The symbol *n(loc)* indicates the number of connections to the specified location so far. It gets its value from the log file. This includes both active and terminated connections. The symbol *req_n(loc)* denotes the requested n^{th} occurrence of connection to the location, which is specified in a rule. The symbol *loc* denotes a boolean expression that evaluates to true if LaseriX finds that the current unprocessed active connection refers to the specified location.

Algorithm for the *at* operation

```
find n(loc);
if (loc) {                          # at the specified location
  if (req_n(loc) != "*") {          # at the specified occurrence
    if (n(loc) == req_n(loc)) {     # matches the requirement
```

```
      return true;
    } else {
      return false;
    }
  } else {                          # at any occurrence
    return true;
  }
} else {                            # not at the specified location
  return false;
}
```

The algorithm is best explained using an example. Consider the following execution rules.

```
at paragon,1 > /dev/null:           # on first time visit only
        cd work/draft/thesis/
        latex thesis.tex
        dvips -o thesis.ps thesis.dvi
at tarma,* > current:               # on every visit
        echo "Welcome to tarma"
```

Consider also the content of a file that logs the user's connection to several machines.

```
ppp0 130.194.224.185 976603220 active
1696 tarma 976603220 stopped
1835 paragon 976603302 active
1847 tarma 976603366 active
1862 insect 976603588 active
1872 insect 976603828 stopped
1899 tarma 976604093 active
1925 paragon 976604879 active
```

At the time the above log is produced, there are four active connections to `paragon` and `tarma` machines, and a connection indicating a PPP session. The rows in the log indicates the sequence of connection activation and deactivation (which is shown by the time stamps as well). When the first connection instance to `paragon` is activated, it matches to the first rule, and the latex file compilation is executed. When the second connection is established, it does not trigger the action because it does not match the occurrence requirement of the rule. On the other hand, every time a connection to `tarma` is started, the welcoming message is written on the current screen.

In a similar manner, the algorithms for the *before* and *at* operation can be defined as follows.

Algorithm for the *before* operation

```
find n(loc);
if (n(loc) < req_n(loc)) {     # it has not reached the required
  return false;                  # number of occurrence
} else {                       # it has not ...
  return true;
}
```

Algorithm for the *after* operation

```
find n(loc);
if (n(loc) > req_n(loc)) {     # it starts matching the required
  return true;                   # number of occurrence
} else {                       # it has not ...
  return false;
}
```

To comply with the Mocha model, evaluation of each clause is performed independently. Therefore it is possible to have concurrent command execution. This happens if more than one clause evaluate to true in an evaluation round. Independent command execution is achieved using threads. For each clause which evaluates to true, a new thread is created. The thread will then perform the execution. In this situation, LaseriX preserves the order of execution *within* a clause, but not between clauses.

One problematic situation for the task scheduler is that the commands it executes due to a clause evaluation has no strong association with the connection process that the execution is based on. The task scheduler only guards the front end (i.e., the execution is initiated upon a connection establishment), but not the back end. The task scheduler knows when the particular connection ends, but it cannot simply kill the associated command execution. This is because the nature of the command may not allow abrupt termination (e.g., interactive mode, compilation, or data transfer). Leaving

the back end open may lead to a situation similar to a race condition where multiple commands are executed one after another without a clear sequence logic. This happens if the life time of the connections and the command execution differs greatly (i.e., the latter has much longer life time). It is the user responsibility to predict such a situation and to act accordingly.

6.4 Implementation Issues

The LaseriX program is implemented using the Perl language. The language is selected for its rich features for text processing and its close association to the Linux environment [SC97, WCO00]. Although still considered experimental, Perl also offers thread programming [Sug98] which is required in executing user commands.

Perl shows its excellence in processing text files and streams. A file can be read simply by defining a file handle and using it to feed an array variable. In the following code snippet,

```
open (RFILE,"myrule.rl") || die "Cannot open rule file";
@lines = <RFILE>;
```

the array `@lines` will contain the content of the file `myrule.rl`. Each array element represents a line in the text, separated by the new line character.

A similar method can also be applied for text streams as a result of a command execution. For example, to capture the textual information of the **ifconfig** command, the following Perl code can be used. Again, the array variable will contain the requested texts.

```
open (IFCFG,"/sbin/ifconfig |") || die "Cannot execute ifconfig";
@configlines = <IFCFG>;
```

Perl also offers a wealthy feature for working with regular expressions. It provides a number of predefined classes for common characters such as

digits, whitespaces, and word characters. Operations on regular expressions, such as pattern matching, substitution, and conversion, can be performed in a concise manner. In the development of LaseriX, this feature is extremely useful for extracting textual information based on specific text patterns. For example, to retrieve remote host information from the output of a **ps ax** command, the following code is used.

```
# run ps ax, then assign the output to the @process_list array
@process_list = qx(ps ax);
# grep lines with 'telnet' or 'ssh'
@session = grep { /(telnet|ssh)[\s]+/ } @process_list;
# process each line that contains 'telnet' or 'ssh'
foreach $i (@session) {
  # split a line using white space as the delimiter
  @l = split(/[\s]+/,$i);
  $session_pid = $l[1];     # PID is the second element
  # remote host is the sixth element, presented in its
  # actual name
  ($rmt_host) = (gethostbyname($l[6]))[0];
}
```

By default, the compiled Perl package shipped in many Linux distributions does not support thread programming. To enable this feature, the package source has to be recompiled with some special compiler switches enabled. Once this has been done, thread programming support can be activated by importing the `Thread` module.

A new thread can be created using the `new` command that takes a function as an argument. When the code is run, the function will be executed in a different thread. Using this mechanism, a separate thread can be created for a clause evaluation which returns a true value. The following code shows how it is done in LaseriX.

```
# parse the clause and arrange the user commands into an array
# called @commands
...
# perform evaluation
...
# the result of the evaluation is stored in $evalresult
if ($evalresult) {
```

```
  $thr = new Thread &execute($host, @commands);
}

# the 'execute' commands looks like the following

sub execute {
  # the @_ is the argument array containing the remote host
  # and the user commands. The host is indicated by @_[0]
  # while the user commands are stored in the i-th elements,
  # where i>0
  for ($i=1; $i<@_; $i++) {
    chomp;
    if ($local eq @_[0]) {
      system(@_[$i]);                    # run it locally
    } else {
      system("ssh ".@_[0]." ".@_[$i]);   # run it remotely
  }
}
```

As command executions are completely independent of each other and from the main thread, there is no need for any synchronisation. Also as previously mentioned, the main thread does not wait for the execution threads to finish as it may need to prepare for the next context state change that triggers a new evaluation. The consequence is that it is not possible to control the outcome of the execution (e.g., performing error handling, and preventing race condition on output when the same display is used by multiple, consecutive threads).

The way remote execution is conducted requires some explanation. LaseriX uses a secure shell connection using **ssh** to execute a command on a remote host. To do this the user must have a valid account on the remote host and must set some configuration so that authentication can take place properly.

The **ssh** program supports RSA-based authentication which is based on public key cryptography [Odl94]. Encryption and decryption are done using separate keys, and it is not possible to derive the decryption key from the encryption key. The **ssh** program requires a user to create a private-public key pair for authentication purposes. The server, or a remote host in

this case, knows the public key, and the private key is known only to the user. Authentication is done in a three-way communication, and by generating and storing the private and public keys in their proper places, the process can be done transparently [CBF+99]. The user will not be asked to supply a password when trying to log into the remote system.

The **ssh** program also allows commands to be specified during its invocation, and this feature is used in LaseriX to execute user commands remotely and transparently. In the program, this feature is activated by a call to the **system** function, which launches a new shell (typically **/bin/sh**) and passes its string argument to be executed by the shell.

6.5 Using LaseriX

This section illustrates how LaseriX can help mobile users organise their tasks to cope with their mobility. The illustration is given through examples, and comparison with the traditional methods is also presented.

A potential use of LaseriX is as a counterpart of the **cron** command. In this case, LaseriX can be used to schedule tasks based on user locations. The feature is useful for mobile users that regularly make some connections, either at the high-level or the low-level. LaseriX allows the execution of tasks that are logically associated with the presence of the user on the location denoted by the connection to be automated.

Consider a network application whose operation is completely dependent on the connection status of the network. For instance, a mobile user wants to launch a web browser when the laptop's modem is active, and to shutdown the browser automatically when it becomes off-line. This can be achieved through the following execution rules. Laserix can then be activated to run as necessary, for example, during the user's travel to offices in different cities.

```
at 130.194.224.46,* > default:
  netscape &;
! (at 130.194.224.46,*) > default:
  killall -9 netscape;
```

The first execution rule states that the Netscape browser is launched on the user's computer every time he or she connects to a remote machine whose IP address is 130.194.224.46 (say the connection is made to a modem server uses the PPP protocol). The second rule states that when the connection is lost, the browser is forcibly killed.

The same effect can be achieved using a conventional approach by creating a wrapper script to activate of both the PPP connection and the browser. On the other end, a similar deactivator wrapper is also created. The deactivator wrapper script periodically checks for the connection status, and if it is down, it terminates the browser. This approach works fine, except that it does not reflect the relationship between the activation of the browser and the state of being connected to the network.

To continue with another example, the user may have to travel because he or she has some tasks that are executable only by travelling to certain places. For instance, a sensitive data transfer can only be executed if the user is physically connected to a secure network. At other times, the user may need to be able to read all email stored in a mail server located in the head office, so it is necessary to fetch all email to the user's laptop wherever he or she goes. At the problem-level perspective, all these tasks are not independent. They form an abstraction of a problem-solving activity that requires user mobility. The traditional method cannot represent this abstraction because it does not capture the temporal property (i.e., the execution sequence) of the tasks.

LaseriX offers a solution by using the location context as the basis of computation in a mobile computing environment. The previous execution rules are rewritten to include the new requirements of the data transfer

and email redirection. Note that the whole configuration file containing the execution rules now describes the user's problem-solving activity. In other words, LaseriX allows task organisation to be captured and represented in a Linux's ordinary file.

```
at 130.194.224.46,* > default:
  netscape &;
! (at 130.194.224.46,*) > default:
  killall -9 netscape;
at 130.194.224.0,1  > default:
  data_transfer;
at office1,* && at office2,* && at office3,* > /dev/null:
  fetchmail mailserver.monash.edu.au;
```

LaseriX also allows a user to do reasoning based on his or her mobility state (refer to Section 3.4.2 for explanation on mobility states). This feature is very useful if there are dependencies among tasks executed on certain hosts. Mobility states can be used to specify the dependencies and automate the execution of the tasks accordingly.

To illustrate the reasoning mechanism, consider that the user from the previous example has to collect some data in all offices before he or she can transfer them to the main server. To do this task, the user has to connect to the data server in each office and run the data collection program, before connecting to the secure network to perform the transfer. This example shows a dependency between two tasks which is reflected by the order of the visit to the associated offices. In distributed computing, there is no way to represent this kind of dependency, which on the other hand is a common situation in mobile computing. LaseriX handles this situation easily using its context operators, as shown below.

```
...
before 130.194.224.0,1 > default:
  local data_collection;
at 130.194.224.0,1  > default:
  data_transfer;
...
```

The key point of the reasoning is the IP address denoting the secure network. It logically connects the two execution rules. The **before** clause captures every connection to any host or network before the user connects to the secure network. The **at** clause represents the situation when the *before* state becomes false, which means the data transfer should be performed. In summary, the IP address allows the two rules in the code precisely model the dependency between the two tasks.

6.6 Discussion

An implementation of the Mocha model in an operating system environment has been presented in this chapter. A summary on the work is given in this section. This section also sums up the experiences and lessons learned from the implementation effort decribed in Chapters 5 and 6. It also explains how Mocha can assist in a development process by complementing the conventional software development tools.

6.6.1 Mobile Applications in an Operating System Environment

The implementation work described in this chapter develops a shell program that allows users to control the execution of user commands based on their locations. Under the definition mentioned in Section 2.4, LaseriX can be called a mobile application since it handles mobility-related issues in its main computation. The prototype interestingly demonstrates that, under the Mocha model, a mobile application does not necessarily have to be built in a programming environment.

The Laserix prototype also demonstrates that a reactive execution model can be implemented on top of standard Linux features. With little help from Perl programming, the reactive mechanism can be built by tap-

ping information provided by Linux commands and log files. This proves two things: (i) mobility programming can be as casual as normal script programming that works on log files, and (ii) that Linux (and Unix in general) is actually ready for mobile computing at the user level, without any need of complex kernel modifications.

The experiment also demonstrates that a mobile application can be developed on a non-conventional programming environment. A Linux (and UNIX) shell is a programming environment with limited programming resources. Shell-based programming is mainly supported by small programs and shell internal commands as the building blocks. In this respect, the role of the Perl program in LaseriX is quite different from that of the Java framework described in Chapter 5. The Perl program is more like a "glue" for the small tool components. In this role, Perl does not provide the notion of a *exclusive* programming environment like Java does. For example, it does not perform the user's commands; it only feeds them to the shell, which actually executes them.

To be really useful for mobile computing, however, Linux has to be able to support more contexts. For some contexts, support have been available. For example, network connection speed can be monitored by calculating incoming IP packet traffic. Developing tools that work with contexts other than location is not that difficult. The development of LaseriX suggests that a generic development pattern does exist. The pattern basically consists of three steps, as shown in the following.

1. Determine how to acquire information about context values. Normally this is achieved by examining log files.

2. Reuse the format of execution rules (see Section 6.3.3) to suit a particular context representation. For example, if the multiple contexts are supported, the syntax has to be modified to reflect this feature.

3. Develop a program that acts as an execution engine. The program reads the required information from some given sources, evaluate the execution rules, then perform command execution accordingly.

The first step in the pattern is the most crucial one, especially if the program makes use of contexts that require non-trivial representation. However, Linux (and Unix) tradition allows this to be implemented by designing a specific format for this purpose. This approach has been used in many Unix tools for their configuration files (e.g., crontab files for **cron**, sendmail.cf file for **sendmail**, or the /etc/host.conf for the DNS mechanism). These configuration files may contain non-trivial representation of operational settings. Given various tools with text-processing capabilities, manipulation of the content of the files should not be difficult.

6.6.2 A Final Note on Developing Mobile Applications

Chapters 5 and 6 described two different courses of implementation of mobile system specifications. Chapter 5 showed that the nature of the work is to help programmers develop mobile applications. In this chapter, the aim of the project is to develop a mobile application. Despite this difference, there is a significant similarity between the two projects. Both focus on the provision of mobility support to mobile applications, and the work is conducted based on the same development approach. From this point of view, the prototypes show the potential of the Mocha modeling paradigm in supporting mobile application development. To summarise the work described in those two chapters, this section presents a general guideline for developing mobile applications based on the Mocha modeling paradigm.

A developer should have a clear description on the mobile system he or she wants to build. A system description should specify the following information.

- Functional tasks that need to be executed at remote locations.
- Relationships among the functional tasks, especially those related to their migration. Some examples include the time requirement for their migration (e.g., they migrate at the same time), the contextual requirements (e.g., they share some common itineraries or they plan a rendezvous), and migration dependency (e.g., the migration of a component is part of the other's).
- Physical environment elements that may affect the performance of the execution of the tasks, and their implication of the presence of their abstraction on the execution of the tasks.

From a software engineering perspective, a model specification describes a set of mobility support requirements. The specification gives an idea about specific aspects related to the mobility that have to be handled by an application. It complements the system specification developed using the traditional system design approach.

Recall that the Mocha paradigm is based on the principle of separation of concerns. The application of this principle is carried out through *task structuring* (see Section 3.3.3). The purpose of task structuring is to identify and categorise system-level elements of a mobile application, and map their relationship in terms of hierarchical structures. This process is similar to system allocation in the system analysis and design stage. However, task structuring specifically aims to separate functional elements from mobility elements.

Conventional system modeling usually does not give much attention to the mobility aspect of a mobile application. As mentioned in Section 4.6, this is due to the design of the modeling tools which cannot capture and represent the essential features of a mobile system. This is the point where Mocha can

help. System modeling can go in parallel from this point. Mocha goes to one direction that focuses on the mobility aspect, while a conventional modeling tool can handle the functional aspects of the application. Once this stage is completed, the model developed using both tools provides a multi-facet specification of the application, with mobility is given a strong emphasis.

Once a model specification is given, the preparation for an implementation can start. The first thing to do is to decide the computing environment where the implementation will take place. The decision should take into account the following aspects.

- *The types of principals of migration.* Mocha does not distinguish mobility support based on the types of mobile entities. However, this factor is significant in selecting the computing environment where the development is taking place. Certain types of principals are better supported if a mobile application is developed in a programming environment. For example, at present, operating systems cannot handle code migration at the user level.

- *Awareness of the physical environment.* To implement a Mocha specification, the development environment has to be able to capture contextual information and represent it in a suitable format. Capturing contextual information (e.g., monitoring context values) largely depends on the features of the development tool that provide interface to the physical environment.

  Trivial contexts such as location are easy to represent, but more complex ones may need a sufficiently powerful abstraction mechanism to express them in an application.

- *Support from the development environment.* Features offered by a development environment significantly determine the style of the mobile application development. The programming framework for Mocha

demonstrates a project with a strong notion of programming, while the development of LaseriX demonstrates a non-programming course of application development.

A programming environment usually provides more flexible development support. The programming language's generic features such as abstraction handling and control of execution flow, combined with language-specific features such multithreaded and distributed programming generally offer more flexibility than that offered by an operating system environment. On the other side, if the developed application works closely with an operating system environment (e.g., the migrated functional components are in the form of operating system tools, as in LaseriX), the operating system environment could be a better option for carrying out the implementation process.

Once an implementation environment has been selected, implementation can be carried out using the rules, features, and restrictions set by the environment. Software components are designed using the programming paradigm provided by the programming environment. In the Java programming environment, for instance, software design and implementation follow the object-oriented paradigm. In the Linux and Unix environment, the application is implemented using UNIX's programming philosophy, which relies on small and generic tools as building blocks to create a more complex program.

To conclude this chapter, this section presents a set of guidance for developing mobile applications based on the Mocha model. It also shows how developers can take advantages of the Mocha approach to complement the conventional modeling approach. This section also concludes the main part of the thesis. The next concluding chapter summarises the thesis and proposes a roadmap to possible research topics that still need to be addressed in the future.

Chapter 7

Conclusion

7.1 Summary of the Thesis

The focus of the research reported in this thesis is on supporting the development of mobile applications. The aim of the research is to achieve a better understanding of this topic by exploring the characteristic of mobile applications and optimising its role in the development of the applications.

Two important aspects that characterise mobile applications have been identified. Firstly, mobile applications need to be aware of the presence of their physical environment. This is achieved by including the abstraction of physical environment elements in the application's computation. Such awareness of physical environment is crucial to the performance of an application. Secondly, mobile applications control the mobility of different kinds of mobile entities that represent people and code. Each type of mobile entity has its own requirements, therefore mobile applications that deal with them have to provide different kinds of support.

To date, the majority of research work in this area has been focusing on a small part of the problem. Mobile application development is often reduced to a programming activity that focuses on a specific aspect of mobility. This partial approach is not adequate for handling complex mobile computing

problems. More importantly, a partial approach prevents developers from having a seamless view of mobile application development. The resulting implementation is not backed by a strong conceptual basis that describes the mobility aspect of an application.

The research addresses the problems at the conceptual level. It concentrates on the modeling aspects and uses them as the starting point for a development process. An appropriate modeling framework is required to create a mobility model that can capture the essential properties of a mobile system. This is an important requirement for any development effort that attempts to provide generic and universal solutions.

A modeling framework called Mocha has been developed to allow developers to specify the behaviour of a mobile application. Mocha highlights the separation of the mobility aspect from the functionality of the application. This is achieved using the *vehicle* metaphor. In this metaphor, a vehicle that represents a mobile entity carries one or more functions that need to be executed remotely. The metaphor treats different types of mobile entities uniformly, allowing generic support to be provided.

The interaction between a mobile entity and its surrounding physical environment is facilitated by *contexts*. A context is a generic abstraction of a physical environment element that surrounds a mobile entity. A context-based mechanism is designed to capture the dynamics of a mobile entity that is caused by its mobility. The mechanism is used to control the execution of the functions of a mobile entity based on the contextual state of the entity.

Mocha also accommodates the design of applications with multiple mobility where mobile entities capable of performing independent migration move concurrently. Interactions between entities are facilitated by *rendezvous* or meetings, which is also realised on top of the context-based mechanism. This allows a rendezvous to be performed on a basis other than location.

Mocha is designed with practical considerations as a priority. It aims to aid developers to carry out application modeling and its implementation. This goal is approached by devising a suitable representation in which a model specification can be written. A specification language, also called Mocha, is designed for this purpose. The novel aspect of the Mocha language is that it retains the abstract nature of the model while at the same time providing assistance towards a model implementation. This is achieved by a set of programming-language-like constructs whose abstract semantics is open to different methods of implementation. The semantics of the constructs allows a specification to be carried out in an abstract manner, while their structure maps easily to the structure of an actual programming language.

The semantics of the language's constructs, however, does not necessarily have to be implemented using a programming language. An implementation prototype running on a Linux shell environment has been built to demonstrate this feature. The prototype is a Linux tool which allows a mobile user to organise the execution of shell commands based on the historical, current, and future aspect of the user's location.

Another prototype has been developed in the Java programming environment. The prototype is a programming framework that allows developers to implement a model specification using Java classes and objects and integrate them with components that build the functionality of the program.

While the prototypes, as research products, are useful by themselves, they also demonstrate the potential of Mocha to supply the missing link connecting the modeling stage and the implementation stage of mobile application development.

7.2 Contributions of the Thesis

This section states the contributions of the thesis, describes how they are achieved, and how they can be used by research and development community.

- Mocha is a modeling framework for mobile application development with a new perspective of mobility. The model highlights the separation of mobility and functionality aspects to promote a generic approach for catering for different types of mobile entities with different mobility requirements. Facilitation of interaction of mobile entities with their physical environment is also a novel aspect of Mocha. The context-based mechanism that represents such an interaction, in fact, can be applied to other types of applications that are not necessarily related to mobility.

- Mocha is designed with application development as the main consideration, and this distinguishes it from other modeling frameworks for mobile computing systems based on theoretical backgrounds. The pragmatic nature is due to the specification language designed for representing a model. The Mocha language has a set of constructs with abstract semantics, and they are structured similar to a programming language. The language's ability to express its abstractions in terms of software development terminologies allows gradual abstraction refinement process in the modeling, design, and implementation stages.

- A Java programming framework has been developed to assist programmers in writing Java programs that implement a model specification. Given the widely supported nature of Java, the framework facilitates developers in creating different types of mobile applications (e.g., mobile agents, network-aware programs, and GPS-based applications) using a uniform approach.

- The research also resulted in a prototype Linux shell extension, called LaseriX, to support mobile users. LaseriX is similar to the **cron** tool, but its operation is based on the user's locations, represented by the user's connections to remote hosts or network. Considering that task automation is an important feature in Linux, LaseriX could be a very useful task automation tool for Linux mobile users.

In summary, the research proposes a new approach for mobile application development that is based on mobility modeling and specification of mobility control. The proposed approach combines a strong theoretical concept for expressing mobility with development-oriented modeling. The result is a new development approach for dealing with the unique aspects of mobile applications.

7.3 Future Work

The research offers some insights on mobile application development. Developers can directly use both the conceptual and pragmatical contributions of the research. However, there is still scope for additional work, particularly in the implementation stage. The non-exhaustive list of possible future work is as follows.

- *The need for a suitable programming language.*
  Section 5.5 presented a critical analysis on Java as the programming language used in Mocha model implementation. The main objections to Java are its inability to map to the semantics of Mocha language constructs and to implement them effectively using basic Java constructs. The semantic transformation problem leads to awkward implementation of some constructs (e.g., context statements and multiple mobility). At present, only ad-hoc solutions can be provided.

For a better solution, a more suitable programming language would be advantageous. The language has to overcome the semantics transformation problem. At the same time it has to offer an easy mechanism for the composition of mobility and functionality components. The design of such a language needs to look into the aspects of mixing the reactive execution model, program composition, and distributed concurrency.

- *Extension of mobile computing support in non-programming environments.*

  The LaseriX system described in Chapter 6 illustrates the potential of using the application of Mocha model in a non-programming environment. LaseriX proves that mobile computing does not have to be supported by developing complex programs. Linux (and Unix in general) has proved to be a suitable environment for developing support for mobile computing. Given the Linux programming philosophy that relies on tools as building blocks (e.g., the concept of *pipe*), it is possible to enhance the features of LaseriX. For example, a proper context monitoring mechanism can be built so that more contexts can be supported. Linux has provided the tools, what is required is simple programming to extend the mechanism.

  A slightly different direction can also be pursued. While LaseriX implements mobility support at the user-level, another alternative is to built the support *inside* the operating system itself. This approach is more complex, but it is more structured. More importantly, the open nature of Linux (i.e., its source code can be obtained, read, and modified freely) allows that kind of experiment to be carried out.

Appendix A

Mocha Language Grammar

This appendix presents the syntax of Mocha specification language. The syntax is extracted from a syntax analyser (parser) program which is developed using the Java Tree Builder (JTB) [WTP00] and JavaCC [Met00] packages.

The syntax is presented in the form of a Java-like file that contains the grammar for Mocha. Some portion of this file is taken from a grammar file included in the JavaCC package [Met00]. Production rules are expressed as Java methods, making use of the EBNF format in their specification. The file has to be preprocessed by the JTB and JavaCC compilers to produce a syntax analyser that takes the form of a Java file. Once the syntax analyser program has been compiled, it can be executed to check the syntax of a Mocha program file given as its argument.

The first part of the grammar deals with lexical tokens, reserved words, and white spaces. This part is shown below.

```
/* WHITE SPACE */

SKIP :
{
  " "
| "\t"
| "\n"
| "\r"
| "\f"
}
```

```
/* COMMENTS */

SPECIAL_TOKEN :
{
  <COMMENT: "#" (~["\n", "\r"])* >
}

/* RESERVED WORDS AND LITERALS */

TOKEN :
{
  < INTEGER: "int" >
| < FLOAT: "float" >
| < STRING: "string" >
| < BOOLEAN: "boolean" >
| < TRUE: "true" >
| < FALSE: "false" >
| < IPADDR: "ipaddr" >
| < VOID: "void" >
| < DUMMY: "_" >
| < SET: "set" >
| < SEQ: "seq" >
| < BAG: "bag" >
| < DYNAMIC: "dynamic" >
| < EXTERNAL: "external" >
| < FUNCTION: "function" >
| < IF: "if" >
| < ELSE: "else" >
| < OF: "of" >
| < WHILE: "while" >
| < FOR: "for" >
| < ON: "on" >
| < USE: "use" >
| < SYNC: "sync" >
| < BEFORE: "before" >
| < AT: "at" >
| < AFTER: "after" >
}

/* LITERALS */

TOKEN :
{
  < INTEGER_LITERAL:   ("-")? (["0"-"9"])+ >
|
  < FLOATING_POINT_LITERAL:
        ("-")?
        ( (["0"-"9"])+ "." (["0"-"9"])* (<EXPONENT>)?
          | "." (["0"-"9"])+ (<EXPONENT>)?
          | (["0"-"9"])+ <EXPONENT>
```

```
        )
    >
|
    < #EXPONENT: [ " e ", " E " ] ([ " + ", " - " ])? ([ " 0 " - " 9 " ])+ >
|
    < STRING_LITERAL:
        " \ " "
        (     (~[ " \ " ", " \ \ ", " \ n ", " \ r " ])
            | (" \ \ "
                ( [ " n ", " t ", " b ", " r ", " f ", " \ \ ", " ' ", " \ " " ]
                | [ " 0 " - " 7 " ] ( [ " 0 " - " 7 " ] )?
                | [ " 0 " - " 3 " ] [ " 0 " - " 7 " ] [ " 0 " - " 7 " ]
                )
            )
        )*
        " \ " "
    >
|
    < BOOLEAN_LITERAL: " true " | " false " >
|
    < IPADDR_LITERAL:
      (
        ([ " A " - " Z " ][ " a " - " z " ][ " 0 " - " 9 " ])+
        ( " . " ([ " A " - " Z " ][ " a " - " z " ][ " 0 " - " 9 " ])+ )*
        |
        [ " 0 " - " 9 " ] ([ " 0 " - " 9 " ])? ([ " 0 " - " 9 " ])? " . "
        [ " 0 " - " 9 " ] ([ " 0 " - " 9 " ])? ([ " 0 " - " 9 " ])? " . "
        [ " 0 " - " 9 " ] ([ " 0 " - " 9 " ])? ([ " 0 " - " 9 " ])? " . "
        [ " 0 " - " 9 " ] ([ " 0 " - " 9 " ])? ([ " 0 " - " 9 " ])?
      )
    >
}

/* IDENTIFIERS */

TOKEN :
{
    < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>)* >
|
    < #LETTER:
        [
         " \ u0024 ",
         " \ u0041 " - " \ u005a ",
         " \ u005f ",
         " \ u0061 " - " \ u007a ",
         " \ u00c0 " - " \ u00d6 ",
         " \ u00d8 " - " \ u00f6 ",
         " \ u00f8 " - " \ u00ff ",
         " \ u0100 " - " \ u1fff ",
         " \ u3040 " - " \ u318f ",
         " \ u3300 " - " \ u337f ",
```

```
        "\u3400"-"\u3d2d",
        "\u4e00"-"\u9fff",
        "\uf900"-"\ufaff"
      ]
  >
|
  <#DIGIT:
      [
        "\u0030"-"\u0039",
        "\u0660"-"\u0669",
        "\u06f0"-"\u06f9",
        "\u0966"-"\u096f",
        "\u09e6"-"\u09ef",
        "\u0a66"-"\u0a6f",
        "\u0ae6"-"\u0aef",
        "\u0b66"-"\u0b6f",
        "\u0be7"-"\u0bef",
        "\u0c66"-"\u0c6f",
        "\u0ce6"-"\u0cef",
        "\u0d66"-"\u0d6f",
        "\u0e50"-"\u0e59",
        "\u0ed0"-"\u0ed9",
        "\u1040"-"\u1049"
      ]
  >
}

/* SEPARATORS */

TOKEN :
{
  <LPAREN: "(">
| <RPAREN: ")">
| <LBRACE: "{">
| <RBRACE: "}">
| <LBRACKET: "[">
| <RBRACKET: "]">
| <SEMICOLON: ";">
| <COLON: ":">
| <COMMA: ",">
| <DOT: ".">
| <DOTDOT: "~">
}

/* OPERATORS */

TOKEN :
{
  <ASSIGN: "=">
| <GT: ">">
| <LT: "<">
```

```
| < BANG: "!" >
| < HOOK: "?" >
| < EQ: "==" >
| < LE: "<=" >
| < GE: ">=" >
| < NE: "!=" >
| < SC_OR: "||" >
| < SC_AND: "&&" >
| < INCR: "++" >
| < DECR: "--" >
| < PLUS: "+" >
| < MINUS: "-" >
| < STAR: "*" >
| < SLASH: "/" >
| < REM: "%" >
| < ASSG: ":=" >
| < DIFF: "|" >
}
```

The next part is the core of the grammar specification. It describes the grammar of Mocha using an abstract syntax tree whose nodes are expressed in terms of Java methods. The tree starts with the `CompilationUnit` definition as its root node. Each syntactic construct of Mocha is defined as a non-terminal node.

```
/*****************************************
 * THE MOCHA LANGUAGE GRAMMAR STARTS HERE *
 *****************************************/


void CompilationUnit() : {} {
  (
    ObjectDefinition() [ <ASSG> Expression() ]   |
    ContextStatement()
  )*
}

void ObjectDefinition() : {} {
  ( <IDENTIFIER> | <DUMMY> ) <COLON>
  Typedef()
  [ <ASSG> Expression() ]
  <SEMICOLON>
}

void Typedef() : {} {
  [ LOOKAHEAD(2) <IDENTIFIER> "=" ]
  (
```

```
    LOOKAHEAD( Type() )
    Type()
    |
    LOOKAHEAD(1, <EXTERNAL> | <FUNCTION> )
    FunctionTypedef()
    |
    LOOKAHEAD( <SET> | <SEQ> | <BAG> )
    CompoundTypedef()
  )
}

void CompoundTypedef() : {} {
  SetTypedef() | SeqTypedef() | BagTypedef()
}

void SetTypedef() : {} {
  <SET> Typedef()
}

void SeqTypedef() : {} {
  <SEQ> Typedef()
}

void BagTypedef() : {} {
  <BAG> BagBody()
}

void BagBody() : {} {
  <LBRACE>
  (
    [ <DYNAMIC> ] <IDENTIFIER> <COLON> Typedef() <SEMICOLON>
  )*
  <RBRACE>
}

void Expression() : {} {
  MultiTermExpression()
}

void SingleTermExpression() : {} {
  <STRING_LITERAL> |
  <BOOLEAN_LITERAL> |
  <IPADDR_LITERAL> |
  <DUMMY> |
  <VOID> |
  <INTEGER_LITERAL> |
  <FLOATING_POINT_LITERAL> |
  LOOKAHEAD(1, <LBRACE>) SetLiteral() |
  LOOKAHEAD(1, <LBRACKET>) SeqLiteral() |
  LOOKAHEAD(1, <LPAREN>) BagLiteral() |
  LOOKAHEAD(2, <IDENTIFIER> <LPAREN> ) FunctionCallExpr() |
```

```
  <IDENTIFIER>void NumericExpression() : {} {
  NumericLiterals()
}

void NumericLiterals() : {} {
  <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL>
}

void PostfixExpression() : {} {
  <IDENTIFIER> ( <DECR> | <INCR> )
}

void PrefixExpression() : {} {
  ( <DECR> | <INCR> ) <IDENTIFIER>
}

void MultiTermExpression() : {} {
  ConditionalOrExpression()
}

void ConditionalOrExpression() : {} {
  ConditionalAndExpression()
  [ <SC_OR> ConditionalOrExpression() ]
}

void ConditionalAndExpression() : {} {
  RelationalExpression()
  [ <SC_AND> ConditionalAndExpression() ]
}

void RelationalExpression() : {} {
  AdditiveExpression()
  [ ComparisonOperator() AdditiveExpression() ]
}

void AdditiveExpression() : {} {
  MultiplicativeExpression()
  [ ( <PLUS> | <MINUS> | <DIFF> ) AdditiveExpression() ]
}

void ComparisonOperator() : {} {
  <EQ> | <GT> | <LT> | <LE> | <GE> | <NE> | <HOOK>
}

void MultiplicativeExpression() : {} {
  SingleTermExpression() ( (<STAR> | <SLASH> | <REM>)
  SingleTermExpression() )*
}


void SetLiteral() : {} {
```

```
  <LBRACE>
  [ LOOKAHEAD(2, RangeElement() <DOTDOT>) RangeExpr() |
    SetSeqElement() ]
  <RBRACE>
}

void SetSeqElement() : {} {
  SingleTermExpression() ( <COMMA> SingleTermExpression() )*
}

void SeqLiteral() : {} {
  <LBRACKET>
  [ LOOKAHEAD(2, RangeElement() <DOTDOT>) RangeExpr() |
    SetSeqElement() ]
  <RBRACKET>
}


void Elements() : {} {
  SingleTermExpression() [ <COMMA> Elements() ]
}

void BagLiteral() : {} {
  <LPAREN> [ Elements() ] <RPAREN>
}

void RangeExpr() : {} {
  RangeElement() <DOTDOT> RangeElement()
}

void RangeElement() : {} {
  <INTEGER_LITERAL> | <BOOLEAN_LITERAL> | <IPADDR_LITERAL>
}

void FunctionCallExpr() : {} {
  <IDENTIFIER> ( <DOT> <IDENTIFIER> )*
  <LPAREN> [ Elements() ] <RPAREN>
}

void FunctionTypedef() : {} {
  InternalFunctionTypeDefinition() |
  ExternalFunctionTypeDefinition()
}


void InternalFunctionTypeDefinition() : {} {
  <FUNCTION>
  <LPAREN> [ FormalParams() ] <RPAREN>
  "->" Type() <LBRACE>  FunctionBody() <RBRACE>
}
```

```
void Type() : {} {
  <INTEGER> | <FLOAT> | <STRING> | <BOOLEAN> | <IPADDR> |
  <VOID> | <IDENTIFIER>
}

void FormalParams() : {} {
  FormalParam() ( <COMMA> FormalParam() )*
}

void FormalParam() : {} {
  <IDENTIFIER> <COLON> Type()
}

void FunctionBody() : {} {
  ( LOOKAHEAD(2, <IDENTIFIER> <COLON>) ObjectDefinition() |
    NonContextualStatement()
  )*
}

void ExternalFunctionTypeDefinition() : {} {
  <EXTERNAL> <FUNCTION> <IDENTIFIER>
  <LPAREN> [ FormalParams() ] <RPAREN>
  "->" Type()
}

void NonContextualStatement() : {} {
  Block()
  |
  ( LOOKAHEAD(2, <IDENTIFIER> <ASSG>) AssignmentStatement() |
    LOOKAHEAD( FunctionCallExpr() ) FunctionCallStatement() |
    ControlStatement()
  )
}

void Block() : {} {
  <LBRACE> ( NonContextualStatement() )* <RBRACE>
}

void AssignmentStatement() : {} {
  <IDENTIFIER> <ASSG> Expression() <SEMICOLON>
}

void FunctionCallStatement() : {} {
  FunctionCallExpr() <SEMICOLON>
}

void ControlStatement() : {} {
  IfStatement() | ForStatement() | WhileStatement()
}
```

```
void IfStatement() : {} {
  <IF> <LPAREN> ConditionalOrExpression() <RPAREN>
  NonContextualStatement()
  [ LOOKAHEAD(2) <ELSE> NonContextualStatement() ]
}

void Literals() : {} {
  <INTEGER_LITERAL> | <FLOATING_POINT_LITERAL> |
  <STRING_LITERAL> | <BOOLEAN_LITERAL> | <IPADDR_LITERAL>
}

void ForStatement() : {} {
  <FOR> <LPAREN> [ ForInit() ] <SEMICOLON>
  [ ForCondition() ] <SEMICOLON>
  [ ForUpdate() ] <RPAREN>
  <LBRACE> ( NonContextualStatement() )* <RBRACE>
}

void ForInit() : {} {
  <IDENTIFIER> <ASSG> NumericExpression()
}

void ForCondition() : {} {
  NumericExpression() ComparisonOperator() NumericExpression()
}

void ForUpdate() : {} {
  LOOKAHEAD(2, <IDENTIFIER> <ASSG>)
  <IDENTIFIER> <ASSG> AdditiveExpression() |
  PostfixExpression()
}

void WhileStatement() : {} {
  <WHILE> <LPAREN> ConditionalOrExpression() <RPAREN>
  <LBRACE> NonContextualStatement() <RBRACE>
}

void ContextStatement() : {} {
  <ON> <IDENTIFIER> ( <COMMA> <IDENTIFIER> )*
  <OF> <IDENTIFIER> ( <COMMA> <IDENTIFIER> )*
  [ <USE> SeqLiteral() ] ContextBody() <SEMICOLON>
}

void ContextBody() : {} {
  <LBRACE> ( ContextClause() )* <RBRACE>
}

void ContextClause() : {} {
  SelectExpr() <COLON>
  <LBRACE> ( NonContextualStatement() )* <RBRACE>
}
```

```
void SelectExpr() : {} {
  MobTerm() ( ( <SC_OR> | <SC_AND> ) MobTerm() )*
}

void MobTerm() : {} {
  [ <BANG> ]
  ( LOOKAHEAD(MobilityStateExpr()) MobilityStateExpr() |
    ConditionalOrExpression()
  )
}

void MobilityStateExpr() : {} {
  <IDENTIFIER> ( <SYNC> <IDENTIFIER> )*
  [ <BANG> ] ( <BEFORE> | <AT> | <AFTER> ) ContextValue()
}

void ContextValue() : {} {
  [ <IDENTIFIER> <SLASH> ]
  ( <STAR> |
    LOOKAHEAD(1) SeqLiteral() |
    Literals()
  )
  [ <SLASH> ( <STAR> | NumericExpression() ) ]
}
```

References

[AAH+96] G. Abowd, C.G. Atkeson, J. Hong, S. Long, R. Kooper, and M. Pinkerton. Cyberguide: A mobile context-aware tour guide. Technical Report 96-27, GVU Center and College of Computing, Georgia Institute of Technology, 1996.

[Abo96] G.D. Abowd. Ubiquitous computing: Research themes and open issues from an application's perspective. Technical Report 96-24, GVU Center and College of Computing, Georgia Institute of Technology, 1996.

[AG97] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley, second edition, 1997.

[Agr01] P.E. Agre. Welcome to the always-on world. *IEEE Spectrum*, 38(1), January 2001.

[ARS97] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 111–130. Springer, 1997.

[BB92] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, 1992.

[BB96] A.V. Bakre and B.R. Badrinath. Indirect transport layer protocols for mobile wireless environment. In T. Imielinsky and H.F. Korth, editors, *Mobile Computing*, pages 229–252. Kluwer Academics, 1996.

[BC97] K. Bharat and L. Cardelli. Migratory applications. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 131–148. Springer, 1997.

[BDD+93] M. Bender, A. Davidson, C. Dong, S. Drach, A. Glenning, K. Jacob, J. Jia, J. Kempf, N. Periakaruppan, G. Snow, and B. Wong. UNIX for nomads: Making UNIX support mobile computing. In *Proceedings of the USENIX Mobile and Location-Independent Computing Symposium*, pages 53–67. USENIX, August 1993.

[Ber94] L. Bergmans. *Composing Concurrent Objects*. PhD thesis, Dept. Computer Science, University of Twente, Netherlands, 1994.

[BG98] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering*, 24(5):376–390, 1998.

[BGW93] A. Barak, S. Guday, and R.G. Wheeler. *The MOSIX Distributed Operating System: Load Balancing for UNIX*, volume 672 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

[BLL92] A. Brickner, M. Litzkow, and M. Livny. Condor technical summary. Technical Report TR1069, Computer Science Department, University of Wisconsin-Madison, January 1992.

[BN84] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[Bog73] J.K. Boggs. IBM remote job entry facility: Generalized subsytem remote job entry facility. Technical Report IBM Technical Disclosure Bulletin 752, IBM, August 1973.

[BPSM98] T. Bray, J. Paoli, and C.M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0 Specification. Technical Report REC-xml-19980210, World-Wide Web Consortium, February 1998.

[BS98] F. Boussinot and J-F. Susini. The SugarCubes toolbox: A reactive Java framework. *Software: Practice and Experiences*, 28(14):1531–1550, 1998.

[BSHB98] J. Bates, M.D. Spiteri, D. Halls, and J. Bacon. Integrating real-world and computer-supported collaboration in the presence of mobility. In *Proceedings of the IEEE 7th Workshop in Enabling Technologies: Infrastructure for Collaborative Enterprises (WET ICE)*, pages 256–261, June 1998.

[Car95] Luca Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, 1995.

[Car99] L. Cardelli. Abstractions for mobile computation. In J. Vitek and C. Jensen, editors, *Secure Internet Programming: Security Issues for Mobile and Distributed Objects*, volume 1603 of *Lecture Notes in Computer Science*, pages 51–94. Springer, 1999.

[CBF+99] A. Campbell, B. Beck, M. Friedl, N. Provos, T. de Raadt, and D. Song. OpenSSH secure shell client. On-line manual,

September 1999. Available as part of the OpenSSH package from http://www.openssh.org/.

[CG97] G. Cugola and C. Ghezzi. CJava: Introducing concurrent objects in Java. In *Proceedings of the 4th International Conference on Object-Oriented Information Systems (OOIS'97)*, pages 504–514, 1997.

[CG98] L. Cardelli and A.D. Gordon. Mobile ambients. In M. Nivat, editor, *Proceedings of Foundations of Software Science and Computational Structures*, pages 140–155. Springer-Verlag, 1998.

[Cha89] A. Chatterjee. Futures: A mechanism for concurrency among objects. In *Proceedings of the 1989 Conference on Supercomputing*, pages 562–567, November 1989.

[CHK97] D. Chess, C. Harrison, and A. Kershenbaum. Mobile agents: Are they a good idea? In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 25–47. Springer-Verlag, 1997.

[CI96] R. Caceres and L. Iftode. Improving the performance of reliable transport protocol in mobile computing environments. In T. Imielinsky and H.F. Korth, editors, *Mobile Computing*, pages 207–228. Kluwer Academics, 1996.

[CM88] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[Dea98] A. Dearle. Toward ubiquitous environments for mobile users. *IEEE Internet Computing*, 2(1):2–12, 1998.

[DNMMS99] P. Dasgupta, N. Narasimhan, L.E. Moser, and P.M. Melliar-Smith. MAgNET: Mobile agents for networked electronic trading. *IEEE Transactions on Knowledge and Data Engineering, Special Issue on Web Applications*, 11(4):509–525, July-August 1999.

[FG96] S. Franklin and A. Graesser. Is it an agent, or just a program ? a taxonomy for autonomous agents. In *Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages*, volume 1193 of *Lecture Notes in Artificial Intelligence*, pages 21–36. Springer-Verlag, 1996.

[FGL+96] C. Fournet, G. Gonthier, J-J. Levy, L. Maranget, and D. Remy. A calculus for mobile agents. In *Proceedings of the Seventh International Conference on Concurrency Theory (Concur'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, 1996.

[FKK96] A.O. Freier, P. Karlton, and P.C. Kocher. The SSL protocol (version 3.0). Technical report, Netscape Communication Corp., March 1996.

[FLM97] T. Finin, Y. Labrou, and J. Mayfield. KQML as an agent communication language. In J. Bradshaw, editor, *Software Agents*, chapter 14. AAAI Press, 1997.

[FPV98] A. Fugetta, G.P. Picco, and G. Vigna. Understanding code mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, May 1998.

[Fre91] D. Freedman. Experience building a process migration subsystem for UNIX. In *Proceedings of the USENIX Winter Conference*, pages 349–354, January 1991.

[FZ94] George H. Forman and John Zahorjan. The challenges of mobile computing. *IEEE Computer*, 27(4):38–47, April 1994.

[GF92] M.R. Genesereth and R.E. Fikes. Knowledge interchange format, version 3.0 reference manual. Technical Report Logic-92-1, Computer Science Department, Stanford University, June 1992.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addiwon-Wesley, 1995.

[GKCR97] R. Gray, D. Kotz, G. Cybenko, and D. Rus. Agent Tcl. In *Mobile Agents: Explanations and Examples*, chapter 4. Manning Publishing, 1997.

[Gon98] L. Gong. Java 2 platform security architecture. Technical report, Sun Microsystems, Inc., February 1998. Available from http://java.sun.com/products/jdk/1.3/docs/guide/security/spec/security-spec.doc.html.

[GTM+97] F. Griffel, T. Tu, M. Münke, M. Merz, W. Lamersdorf, and M. Mira da Silva. Electronic contract negotiation as an application niche for mobile agents. In *Proceedings of the First International Workshop on Enterprise Distributed Object Computing*, pages 354–365, 1997.

[Gup98] V. Gupta. *Solaris Mobile IP: Design and Implementation*. Sun Microsystems, February 1998.

[Hen88] C. Hendrick. Routing information protocol. Technical Report RFC 1058, The Internet Engineering Task Force, June 1988. Available from http://www.ietf.org/.

[Hoa74] C.A.R. Hoare. Monitors: An operating system structuring concept. *Communication of the ACM*, 17(10):549–557, October 1974.

[HP87] D.J. Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification*. Dorset House, 1987.

[HPRP92] J.S. Heidemann, T.W. Page, R.G.Guy, and G.J. Popek. Primarily disconnected operation: Experiences with Ficus. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 2–5, November 1992.

[HSB99] L. Hazard, J-F Susini, and F. Boussinot. The Junior reactive kernel. Technical Report 3732, INRIA, July 1999.

[HTK98] J. Herstad, D.V. Thanh, and S. Kristoffersen. Wireless Markup Language as a framework for interaction with mobile computing and communication devices. In *Proceedings of the First Workshop on Human-Computer Interaction with Mobile Devices*, pages 89–97, May 1998.

[IDJ91] J. Ioannisdis, D. Duchamp, and G.Q. Maguire Jr. IP-based protocols for mobile internetworking. In *Proceedings of the Symposium on Communication Architectures and Protocols (SIGCOMM'91)*, pages 235–245, September 1991.

[IK96] T. Imielinski and H.F. Korth. Introduction to mobile computing. In T. Imielinsky and H.F. Korth, editors, *Mobile Computing*, pages 1–43. Kluwer Academics, 1996.

[ISI81] University of Southern California Information Sciences Institute. Transmission control protocol. Technical Report RFC 793, The Internet Engineering Task Force, September 1981. Available from http://www.ietf.org/.

[JvRS95] D. Johansen, R. van Renesse, and F.B. Schneider. An introduction to the TACOMA distributed system. Technical Report 95-23, Institute of Mathematical and Physical Sciences, University of Tromsø, June 1995.

[KI98] K. Kawachiya and H. Ishikawa. NaviPoint: An input device for mobile information browsing. In *Proceedings of the ACM CHI Conference on Human Factor in Computing Systems*, pages 1–8, 1998.

[Kla00] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corporation, January 2000.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C.V. Lopes, J-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer-Verlag, 1997.

[Knu95] P. Knudsen. Comparing two distributed computing paradigms: A performance case study. Master's thesis, University of Tromsø, 1995.

[KS92] J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[LK98] C.V. Lopes and G. Kiczales. Recent development in AspectJ. In *Proceedings of the ECOOP'98 Workshop on Aspect-*

Oriented Programming, 1998. Available from http://wwwtrese.cs.utwente.nl/aop-ecoop98/.

[LO98] D.B. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglet*. Addison-Wesley, 1998.

[LSP+97] P.D. Le, B. Srinivasan, R. Price, S. Mohammed, and H.P. Le. Abstract ticket engine to support mobile users. In *Proceedings of the 2nd Australian Workshop on Mobile Computing, Databases, and Applications*, pages 102–107, 1997.

[Mar99] W. Mark. Turning pervasive computing into mediated spaces. *IBM Systems Journal*, 38(4):677–692, 1999.

[Met00] Metamata and Sun Microsystems. *JavaCC version 2.0*, October 2000. Available from http://www.metamata.com/javacc/.

[Mey97] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[MGR+99] N. Minar, M. Gray, O. Roup, R. Krikorian, and P. Maes. Hive: Distributed agents for networking things. In *Proceedings of the First International Symposium on Agent Systems and Applications and Third International Symposium on Mobile Agents (ASA/MA'99)*, pages 118–129, 1999.

[Mic98] Sun Microsystems. Java Remote Method Invocation (RMI) Specification, 1998. Available from http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html.

[Mic00] Sun Microsystems. Java 2 platform micro edition (J2ME) technology for creating mobile devices, May 2000. White Paper. Available from http://java.sun.com/products/kvm/.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts 1 and 2. *Information and Computation*, 100(1):1–77, 1992.

[MY93] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[NLS+00] L.E. Nugroho, S.W. Loke, B. Srinivasan, A.S.M. Sajeev, and I.K. Ismail. A context-based model for programming mobility. In *Proceedings of the Second International Workshop on Information Integration and Web-Based Applications and Services (IIWAS'2000)*, pages 127–141, September 2000.

[NPS95] B.D. Noble, M. Price, and M. Satyanarayanan. A programming interface for application-aware adaptation in mobile computing. In *Proceedings of the 2nd USENIX Symposium on Mobile and Location-Independent Computing*, pages 57–66. USENIX, 1995.

[NS99] L.E. Nugroho and A.S.M. Sajeev. Java4P: Java with high-level concurrency constructs. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms, and Networks (I-SPAN'99)*, pages 328–333. IEEE Computer Society, 1999.

[NS00] L.E. Nugroho and B. Srinivasan. Separation of concerns in mobile object programs: The case of supporting mobility and concurrency. In *Advances in Mobile Agents Systems Research*, volume 1, pages 23–27. IIAS, 2000.

[NT94] B.C. Neuman and T. Ts'o. Kerberos: An authentication service for computer networks. *IEEE Communications*, 32(9):33–38, September 1994.

[Nut94] M. Nuttall. A brief survey of systems providing process or object migration facilities. *Operating Systems Review*, 24(4):64–80, October 1994.

[Obj99] ObjectSpace. *Voyager ORB Developer Guide*, 1999. Available from http://www.objectspace.com/.

[Obj00] The Object Management Group. *OMG Event Service Specification v1.0*, formal/2000-06-15 edition, June 2000.

[Odl94] A.M. Odlyzko. Public key cryptography. *AT&T Technical Journal*, 73(5), 1994.

[Ous94] J. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Par72] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[PB96] C.E. Perkins and P. Bhagwat. Routing over multi-hop wireless network of mobile computers. In T. Imielinsky and H.F. Korth, editors, *Mobile Computing*, pages 183–205. Kluwer Academics, 1996.

[Per97] C.E. Perkins. Mobile IP. *IEEE Communications*, 35(5):84–99, May 1997.

[Per98] C.E. Perkins. Mobile networking through Mobile IP. *IEEE Internet Computing*, 2(1):58–69, Jan–Feb 1998.

[PH98] C. Pfisterer and F. Hohl. *The Mole Cookbook (How to Program a Mole Agent.* Institute of Parallel and Distributed High-Performance Systems, University of Stuttgart, 1998.

[Pic98] G.P. Picco. μCode: A lightweight and flexible mobile code toolkit. In K. Rothermel and F. Hohl, editors, *Proceedings of the Second International Workshop on Mobile Agents (MA'98)*, volume 1477 of *Lecture Notes on Computer Science*, pages 160–171, 1998.

[Pre97] R. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fourth edition, 1997.

[PRWS98] T.F. La Porta, R. Ramjee, T. Woo, and K.K. Sabnani. Experiences with network-based user agents for mobile applications. *Mobile Networks and Applications*, 3(2):123–141, August 1998.

[QPM+93] J. Quemada, L.F. Pires, J.A. Manas, A. Azcorra, and T. Robles. Introduction to LOTOS. In K.J. Turner, editor, *Using Formal Description Techniques.* Wiley, 1993.

[Red97] F.E. Redmond. *DCOM: Microsoft Distributed Component Object Model.* IDG Books, 1997.

[RHB+97] P.V. Roy, S. Haridi, P. Brand, G. Smolka, M. Mehl, and R. Scheidhauer. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):801–851, September 1997.

[Ric95] T. Richardson. Teleporting: Mobile X session. *The X Resource*, 13(1):133–140, 1995.

[RJB99] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[RMP97] G-C Roman, P.J. McCann, and J.Y. Plun. Mobile UNITY: Reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, July 1997.

[SAG+93] B.N. Schilit, N. Adams, R. Gold, M. Tso, and R. Want. The PARCTAB mobile computing system. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, pages 34–39. IEEE, 1993.

[Sat96] M. Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 1–7, 1996.

[SBH97] M. Strasser, J. Baumann, and F. Hohl. Mole: A Java-based mobile agent system. In M. Mühlhäuser, editor, *Special Issues in Object-Oriented Programming, Workshop Reader of the 10th European Conference on Object-Oriented Programming (ECOOP'96)*, pages 301–306, 1997.

[SC97] R.L. Schwartz and T. Christiansen. *Learning Perl*. O'Reilly, 2 edition, 1997.

[SG90] J.W. Stamos and D.K. Gifford. Implementing remote evaluation. *IEEE Transactions on Software Engineering*, 16(7):710–722, July 1990.

[SG97] J.L. Sibert and M. Gokturk. A finger-mounted, direct pointing device for mobile computing. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 41–42, 1997.

[SG99] A. Silberschatz and P.B. Galvin. *Operating System Concepts*. John Wiley and Sons, fifth edition, 1999.

[Sie96] J. Siegel. *CORBA Fundamentals and Programming*. John Wiley and Sons, 1996.

[SNS88] J.G. Steiner, B.C. Neuman, and J.I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the Winter 1988 USENIX Conference*, pages 191–201, February 1988.

[SNS00] A.S.M. Sajeev, L.E. Nugroho, and B. Srinivasan. A framework for programming mobile applications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2000)*, pages 2043–2048, June 2000.

[Spi92] M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall International, 2 edition, 1992.

[SS80] M. Schwartz and T. Stern. Routing techniques used in computer communication networks. *IEEE Transactions on Communications*, COM-28:539–552, April 1980.

[Sta86] J.W. Stamos. *Remote Evaluation*. PhD thesis, Massachusetts Institute of Technology, January 1986.

[Str97] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 1997.

[Sug98] D. Sugalski. Threads. *The Perl Journal*, 3(2), 1998.

[Sun99] Sun Microsystems. *Java 2 SDK, Standard Edition Documentation, Version 1.2.2-001*, 1999. Available from http://java.sun.com/products/jdk/1.2/docs/index.html.

[Sys85] Adobe Systems. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[TA98] K. Taguchi and K. Araki. A calculus based on the Agent-Place model. In *Proceedings of the International Conference on Formal Engineering Methods*, pages 56–63, 1998.

[Ten00] D. Tennenhouse. Proactive computing. *Communications of the ACM*, 43(5):43–50, May 2000.

[TKV+98] A. Tripathi, N. Karnik, M. Vora, T. Ahmed, and R.D. Singh. Ajanta: A mobile agent programming system. Technical Report TR98-016, Department of Computer Science, University of Minnesota, April 1998.

[TO00] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Research, 2000. Available from http://www.alphaworks.ibm.com/tech/hyperj.

[TOHJ99] P. Tarr, H. Ossher, W. Harrison, and S.M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering (ICSE'99)*, pages 107–119, 1999.

[VB94] G.M. Voelker and B.N. Bershad. Mobisaic: An information system for mobile wireless computing environment. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 185–190. IEEE CS Press, 1994.

[vEVD89] P.H.J. van Eijk, C.A. Vissers, and M. Diaz, editors. *The Formal Description Technique LOTOS*. Elsevier Science Publisher, 1989.

[vR00] G. van Rossum. *Python Reference Manual: Release 2.0*, October 2000. Available from http://www.python.org.

[VST97] J. Vitek, M. Serrano, and D. Thanos. Security and communication in mobile object systems. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 177–199. Springer, 1997.

[WAP98] The WAP Forum. *WAP Architecture Specification*, version 30-April-1998 edition, 1998.

[WAP00] The WAP Forum. *WAP WML Specification*, version 1.3 edition, February 2000.

[WB98] G. Welling and B.R. Badrinath. An architecture for exporting environment awareness to mobile computing applications. *IEEE Transactions on Software Engineering*, 24(5):391–400, 1998.

[WBWW90] R. Wirfs-Brock, B. Wilkerson, and L. Weiner. *Designing Object-Oriented Software*. Prentice-Hall, 1990.

[WCO00] L. Wall, T. Christiansen, and J. Orwant. *Programming Perl*. O'Reilly, 3 edition, 2000.

[Wei93] M. Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):75–84, July 1993.

[WHFG92] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location systems. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[Whi94] J.E. White. Telescript technology: The foundation for the electronic marketplace (white paper), 1994.

[WJH97] A. Ward, A. Jones, and A. Hopper. A new location technique for the active office. *IEEE Personal Communications*, 4(5):42–47, October 1997.

[WR96] C.D. Wilcox and G-C. Roman. Reasoning about places, times, and actions in the presence of mobility. *IEEE Transactions on Software Engineering*, 22(4):225–247, April 1996.

[WRB+97] K.R. Wood, T. Richardson, F. Bennett, A. Harter, and A. Hopper. Global teleporting with Java: Toward ubiquitous personalized computing. *IEEE Computer*, 30(2):53–59, February 1997.

[WTP00] W. Wang, K. Tao, and J. Palsberg. *JTB Documentation, ver. 1.2.2.* Purdue University, May 2000. Available from http://www.cs.purdue.edu/jtb/index.html.